



MALLA REDDY INSTITUTE OF TECHNOLOGY & SCIENCE

(SPONSORED BY MALLA REDDY EDUCATIONAL SOCIETY)

Permanently Affiliated to JNTUH & Approved by AICTE, New Delhi

NBA Accredited Institution, An ISO 9001:2015 Certified, Approved by UK Accreditation Centre
Granted Status of 2(f) & 12(b) under UGC Act. 1956, Govt. of India.



PYTHON PROGRAMMING

COURSE FILE



DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

(2022-2023)

COURSE FILE

SUBJECT: PYTHON PROGRAMMING

ACADEMIC YEAR: 2022-2023.

REGULATION: R18

NAME OF THE FACULTY: G. SANDHYA

DEPARTMENT: CSE

YEAR & SECTION: II CSE(CS),CSE(DS),CSE(AI&ML)

SUBJECT CODE: CS311PC

MRITS

PYTHON PROGRAMMING

1. Syllabus Copy

CSC11PC: Python Programming

II Year B.Tech. CSE(CS) I-Sem

L T P C

3 0 0 3

Prerequisites

1. A course on "Python Programming"

Course Objectives

- Learn Syntax and Semantics and create Functions in Python.
- Handle Strings and Files in Python.
- Understand Lists, Dictionaries and Regular expressions in Python.
- Implement Object Oriented Programming concepts in Python.
- Build Web Services and introduction to Network and Database Programming in Python.

Course Outcomes

- Examine Python syntax and semantics and be fluent in the use of Python flow control and functions.
- Demonstrate proficiency in handling Strings and File Systems.
- Create, run and manipulate Python Programs using core data structures like Lists, Dictionaries and use Regular Expressions.
- Interpret the concepts of Object-Oriented Programming as used in Python.
- Implement exemplary applications related to Network Programming, Web Services and Databases in Python.

UNIT - I

Python Basics, Objects- Python Objects, Standard Types, Other Built-in Types, Internal Types, Standard Type Operators, Standard Type Built-in Functions, Categorizing the Standard Types, Unsupported Types Numbers - Introduction to Numbers, Integers, Floating Point Real Numbers, Complex Numbers, Operators, Built-in Functions, Related Modules Sequences - Strings, Lists, and Tuples, Mapping and Set Types

UNIT - II

FILES: File Objects, File Built-in Function [open()], File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments, File System, File Execution, Persistent Storage Modules, Related Modules Exceptions: Exceptions in Python, Detecting and Handling Exceptions, Context Management, *Exceptions as Strings, Raising Exceptions, Assertions, Standard Exceptions,

PYTHON PROGRAMMING

*Creating Exceptions, Why Exceptions (Now)?, Why Exceptions at All?, Exceptions and the sys Module, Related Modules Modules: Modules and Files, Namespaces, Importing Modules, Importing Module Attributes, Module Built-in Functions, Packages, Other Features of Modules

UNIT - III

Regular Expressions: Introduction, Special Symbols and Characters, Res and Python Multithreaded Programming: Introduction, Threads and Processes, Python, Threads, and the Global Interpreter Lock, Thread Module, Threading Module, Related Modules

UNIT - IV

GUI Programming: Introduction, Tkinter and Python Programming, Brief Tour of Other GUIs, Related Modules and Other GUIs WEB Programming: Introduction, Web Surfing with Python, Creating Simple Web Clients, Advanced Web Clients, CGI-Helping Servers Process Client Data, Building CGI Application Advanced CGI, Web (HTTP) Servers

UNIT - V

Database Programming: Introduction, Python Database Application Programmer's Interface (DB-API), Object Relational Managers (ORMs), Related Modules

TEXT BOOK:

1. Core Python Programming, Wesley J. Chun, Second Edition, Pearson.

REFERENCE BOOKS:

1. Think Python, Allen Downey, Green Tea Press
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson
4. Learning Python, Mark Lutz, O'Really



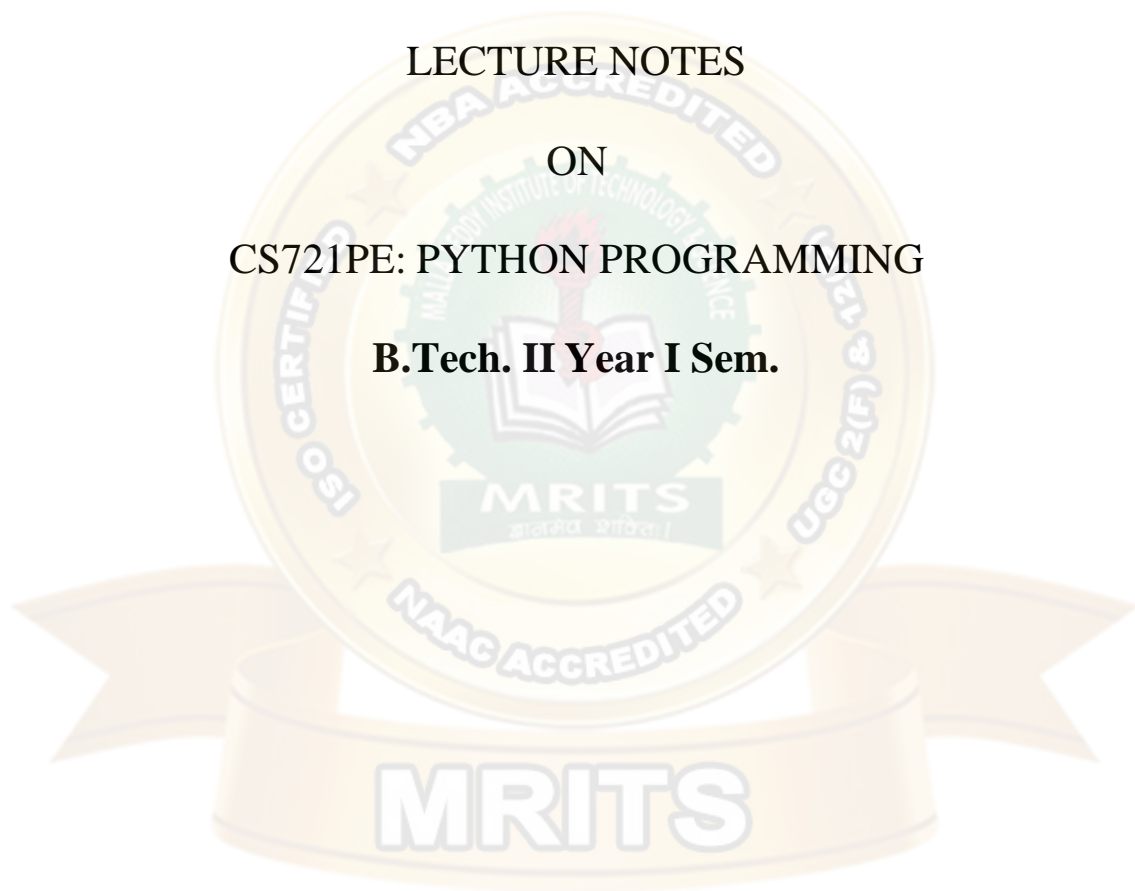
MALLA REDDY INSTITUTE OF TECHNOLOGY AND SCIENCE

LECTURE NOTES

ON

CS721PE: PYTHON PROGRAMMING

B.Tech. II Year I Sem.



1 UNIT

1) Explain about python basics

Python Basics:

- a) Statements and Syntax
- b) Variable Assignment
- c) Identifiers and Keywords
- d) Basic Style Guidelines
- e) Memory Management
- f) First Python Programs

a) Statements and Syntax :

Some rules and certain symbols are used with regard to statements in Python:

- i) **Comments (#)**: Hash mark (#) indicates Python comments
- ii) **Continuation (\)**: NEWLINE (\n) is the standard line separator (one statement per line)
Backslash (\) continues a line
- iii) **Multiple Statement Groups as Suites (:)**: Colon (:) separates a header line from its suite
Statements (code blocks) grouped as suites
- iv) **Suites Delimited via Indentation**: styles
- v) **Multiple Statements on a Single Line (;)**: Semicolon (;) joins two statements on a line
- iv) **Modules**: Python files organized as modules

b) Variable Assignment :

i) Assignment Operator

The equal sign (=) is the main Python assignment operator. (The others are augmented assignment operator [see next section].)

Ex:

```
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [3.14e10, '2nd elmt of a list', 8.82-4.371j]
```

ii) Augmented Assignment:

- Beginning in Python 2.0, the equal sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as augmented assignment, statements such as

`x = x + 1`

- ... can now be written as ...
- `x += 1`

iii) Multiple Assignment

```
x = y = z = 1
```

```
>>>x
```

```
1
```

```
>>>y
```

```
1
```

```
>>>x
```

```
1
```

- In the above example, an integer object (with the value 1) is created, and x , y , and z are all assigned
- the same reference to that object. This is the process of assigning a single object to multiple variables.

c) Identifiers and Keywords :

Identifiers are the set of valid strings that are allowed as names in a computer language. From this all-encompassing list, we segregate out those that are keywords, names that form a construct of the language. Such identifiers are reserved words that may not be used for any other purpose, or else a syntax error (`SyntaxError` exception) will occur.

Python also has an additional set of identifiers known as built-ins, and although they are not reserved words, use of these special names is not recommended.

i) Valid Python Identifiers:

The rules for Python identifier strings are like most other high-level programming languages that come

from the C world:

- First character must be a letter or underscore (`_`)
- Any additional characters can be alphanumeric or underscore
- Case-sensitive

ii) Keywords

Python's keywords Generally, the keywords in any language should remain relatively stable, but should things ever change (as Python is a growing and evolving language), a list of keywords as well as an `iskeyword()` function are available in the keyword module.

iii) Built-ins:

- In addition to keywords, Python has a set of "built-in" names available at any level of Python code that are either set and/or used by the interpreter. Although not keywords, built-ins should be treated as "reserved for the system" and not used for any other purpose. However, some circumstances may call for overriding (aka redefining, replacing) them. Python does not support overloading of identifiers, so only one name "binding" may exist at any given time.

- We can also tell advanced readers that built-ins are members of the `__builtins__` module, which is automatically imported by the interpreter before your program begins or before you are given the `>>>` prompt in the interactive interpreter. Treat them like global variables that are available at any level of Python code.

iv) Special Underscore Identifiers:

- Python designates (even more) special variables with underscores both prefixed and suffixed. We will also discover later that some are quite useful to the programmer while others are unknown or useless. Here is a summary of the special underscore usage in Python:

- `__xxx` Do not import with 'from module import *'
- `__xxx__` System-defined name
- `__xxx` Request private name mangling in classes

d) Basic Style Guidelines:

i) Comments: You do not need to be reminded that comments are useful both to you and those who come after you. This is especially true for code that has been untouched by man (or woman) for a time (that means several months in software development time). Comments should not be absent, nor should there be novellas. Keep the comments explanatory, clear, short, and concise, but get them in there. In the end, it saves time and energy for everyone. Above all, make sure they stay accurate!

ii) Documentation: Python also provides a mechanism whereby documentation strings can be retrieved dynamically through the `__doc__` special variable. The first unassigned string in a module, class declaration, or function declaration can be accessed using the attribute `obj.__doc__` where `obj` is the module, class, or function name. This works during runtime too!

iii) Indentation: Since indentation plays a major role, you will have to decide on a spacing style that is easy to read as well as the least confusing. Common sense also plays a role in choosing

how many spaces or columns to indent. 1 or 2 Probably not enough; difficult to determine which block of code statements belong to 8 to 10 May be too many; code that has many embedded levels will wrap around, causing the source to be difficult to read Four spaces is very popular, not to mention being the preferred choice of Python's creator. Five and six are not bad, but text editors usually do not use these settings, so they are not as commonly used. Three and seven are borderline cases.

e) **Memory Management :**

So far you have seen a large number of Python code samples. We are going to cover a few more details about variables and memory management in this section, including:

- Variables not declared ahead of time
- Variable types not declared
- No memory management on programmers' part
- Variable names can be "recycled"
- del statement allows for explicit "deallocation"

i) **Variable Declarations (or Lack Thereof):**

In most compiled languages, variables must be declared before they are used. In fact, C is even more restrictive: variables have to be declared at the beginning of a code block and before any statements are given. Other languages, like C++ and Java, allow "on-the-fly" declarations, i.e., those which occur in the middle of a body of code but these name and type declarations are still required before the variables can be used. In Python, there are no explicit variable declarations. Variables are "declared" on first assignment. Like most languages, however, variables cannot be accessed until they are (created and) assigned:

```
>>> a
```

```
Traceback (innermost last):
```

```
File "<stdin>", line 1, in ?
```

```
NameError: a
```

Once a variable has been assigned, you can access it by using its name:

```
>>> x = 4
```

```
>>> y = 'this is a string'
```

```
>>> x
```

```
4
```

```
>>> y
```

```
'this is a string'
```

ii) **Dynamic Typing:**

Another observation, in addition to lack of variable declaration, is the lack of type specification. In Python, the type and memory space for an object are determined and allocated at runtime. Although code is byte-compiled, Python is still an interpreted language. On creation that is, on assignment the interpreter creates an object whose type is dictated by the syntax that is used for the operand on the right-hand side of an assignment. After the object is created, a reference to that object is assigned to the variable on the left-hand side of the assignment.

iii) Memory Allocation: As responsible programmers, we are aware that when allocating memory space for variables, we are borrowing system resources, and eventually, we will have to return that which we borrowed back to the system. Python simplifies application writing because the complexities of memory management have been pushed down to the interpreter. The belief is that you should be using Python to solve problems with and not have to worry about lower-level issues that are not directly related to your solution.

iv) Reference Counting:

To keep track of objects in memory, Python uses the simple technique of reference counting. This means that internally, Python keeps track of all objects in use and how many interested parties there are for any particular object. You can think of it as simple as card-counting while playing the card game blackjack or 21. An internal tracking variable, called a reference counter, keeps track of how many references are being made to each object, called a refcount for short. When an **object is created**, a reference is made to that object, and when it is no longer needed, i.e., when an object's refcount goes down to zero, it is garbage-collected. (This is not 100 percent true, but pretend it is for now.)

Let us say we make the following declarations:

```
x = 3.14
```

```
y = x
```

The statement `x = 3.14` allocates a floating point number (float) object and assigns a reference `x` to it. `x` is the first reference, hence setting that object's refcount to one. The statement `y = x` creates an alias `y`, which "points to" the same integer object as `x`. A new object is not created for `y`.

v) Garbage Collection: Memory that is no longer being used is reclaimed by the system using a mechanism known as garbage collection.

f) First Python Programs:

File Create (`makeTextFile.py`)

This application prompts the user for a (nonexistent) filename, then has the user enter each line of that file (one at a time). Finally, it writes the entire text file to disk.

```
#!/usr/bin/env python
```

```

'makeTextFile.py -- create text file'
import os
ls = os.linesep
# get filename
while True:
if os.path.exists(fname):
print "ERROR: '%s' already exists" % fname
else:
break
# get file content (text) lines
all = []
print "\nEnter lines ('.' by itself to quit).\n"
# loop until user terminates input
while True:
entry = raw_input('> ')
if entry == '.':
break
else:
all.append(entry)
# write lines to file with proper line-ending
fobj = open(fname, 'w')
fobj.writelines(['%s%s' % (x, ls) for x in all])
fobj.close()
print 'DONE!'

```

2) Explain about Python Objects in python

- Python Objects
- Standard Types
- Other Built-in Types
- Internal Types
- Standard Type Operators a)Value Comparison b)Object Identity Comparison c)Boolean
- Standard Type Built-in Functions
- Categorizing the Standard Types
- Unsupported Types

A) Python Objects:

Python uses the object model abstraction for data storage. Any construct that contains any type of value is an object. Although Python is classified as an "object-oriented programming (OOP) language," OOP is not required to create perfectly working Python applications.

All Python objects have the following three characteristics: an identity, a type, and a value.

- **IDENTITY** Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the `id()` built-in function (BIF). This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all.
- **TYPE** An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the `type()` BIF to reveal the type of a Python object. Since types are also objects in Python (did we mention that Python was object-oriented?), `type()` actually returns an object to you rather than a simple literal.
- **VALUE** Data item that is represented by an object.

Object Attributes: Certain Python objects have attributes, data values or executable code such as methods, associated with them. Attributes are accessed in the dotted attribute notation, which includes the name of the associated object. The most familiar attributes are functions and methods, but some Python types have data attributes associated with them. Objects with data attributes include (but are not limited to): classes, class instances, modules, complex numbers, and files.

B) Standard Types:

- **Numbers** (separate subtypes; three are integer types)

a) Integer

- Boolean

- Long integer

b) Floating point real number

c) Complex number

ii) **String**

iii) **List**

iv) **Tuple**

v) **Dictionary**

We will also refer to standard types as "primitive data types" in this text because these types represent the primitive data types that Python provides.

C) Other Built-in Types

- Type
- Null object (None)
- File
- Set/Frozenset
- Function/Method

- Module
- Class

a) Type Objects and the type Type Object:

```
>>> type(42)
<type 'int'>
```

b) Null object (None):

- None
- False (Boolean)
- Any numeric zero

D) Internal Types

- Code
- Frame
- Traceback
- Slice
- Ellipsis
- Xrange

i) Code Objects: Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the compile() BIF. Such objects are appropriate for execution by either exec or by the eval () **BIF**

ii) Frame Objects: These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment.

iii) Traceback Objects:

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

```
Traceback (innermost last):
File "<stdin>", line N?, in ???
ErrorName: error reason
```

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

iv) Slice Objects:

Slice objects are created using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include stride indexing, multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is sequence [start1 : end1 , start2 : end2] , or using the ellipsis, sequence [..., start1 :

end] . Slice objects can also be generated by the slice() BIF. Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of sequence[starting_index : ending_index : stride] .

```
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
>>> foostr[::-2]
'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1]
['abc', 342.23, 'xba', 123]
```

v) Ellipsis Objects:

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (...). Like the Null object None , ellipsis objects also have a single name, Ellipsis , and have a Boolean TRue value at all times.

iv)XRRange Objects:

Range objects are created by the BIF xrange() , a sibling of the range() BIF, and used when memory is limited and when range() generates an unusually large data set. You can find out more about range() and xrange().

E) Standard Type Operators:

i) Object Value Comparison

Standard Type Value Comparison Operators

expr1 < expr2 expr1 is less than expr2

expr1 > expr2 expr1 is greater than expr2

expr1 <= expr2 expr1 is less than or equal to expr2

expr1 >= expr2 expr1 is greater than or equal to expr2

expr1 == expr2 expr1 is equal to expr2

expr1 != expr2 expr1 is not equal to expr2 (C-style)

expr1 <> expr2

[a]

expr1 is not equal to expr2 (ABC/Pascal-style)

ii) Object Identity Comparison:

In addition to value comparisons, Python also supports the notion of directly comparing objects themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

Example 1: foo1 and foo2 reference the same object

```
foo1 = foo2 = 4.3
```

When you look at this statement from the value point of view, it appears that you are performing a multiple assignment and assigning the numeric value of 4.3 to both the foo1 and foo2 variables.

iii) Boolean:

Expressions may be linked together or negated using the Boolean logical operators and, or, and not, all of which are Python keywords. These Boolean operations are in highest-to-lowest order of precedence.

Standard Type Boolean Operators

Operator	Function
not expr	Logical NOT of expr (negation)
expr1 and expr2	Logical AND of expr1 and expr2 (conjunction)
expr1 or expr2	Logical OR of expr1 and expr2 (disjunction)

```
>>> x, y = 3.1415926536, -1024
>>> x < 5.0
True
>>> not (x < 5.0)
False
>>> (x < 5.0) or (y > 2.718281828)
True
>>> (x < 5.0) and (y > 2.718281828)
False
>>> not (x is y)
True
```

F) Standard Type Built-in Functions:

Along with generic operators, which we have just seen, Python also provides some BIFs that can be applied to all the basic object types: cmp(), repr(), str(), type(), and the single reverse or back quotes (``) operator, which is functionally equivalent to repr().

Function	Operation
• cmp(obj1, obj2)	Compares obj1 and obj2, returns integer i where: i < 0 if obj1 < obj2 i > 0 if obj1 > obj2 i == 0 if obj1 == obj2
• repr(obj) or `obj`	Returns evaluable string representation of obj
• str(obj)	Returns printable string representation of obj

i) type()

type(object)

type() takes an object and returns its type. The return value is a type object.

```
>>> type(4)                # int type
<type 'int'>
>>>
>>> type('Hello World!')  # string type
<type 'string'>
>>>
>>> type(type(4))          # type type
<type 'type'>
```

ii) **cmp()**: The **cmp()** BIF CoMPares two objects, say, **obj1** and **obj2**, and returns a negative number (integer) if **obj1** is less than **obj2**, a positive number if **obj1** is greater than **obj2**, and zero if **obj1** is equal to **obj2**.

```
>>>a, b = -4, 12
>>>cmp(a,b)
-1
>>>cmp(b,a)
1
>>>b = -4
>>>cmp(a,b)
0
>>>
>>>a, b = 'abc', 'xyz'
>>>cmp(a,b)
>>>cmp(b,a)
-23
>>>b = 'abc'
>>>cmp(a,b)
0
```

iii) **str()** and **repr()** (and **``** Operator):

The **str()** STRing and **repr()** REPResentation BIFs or the single back or reverse quote operator (**``**) come in very handy if the need arises to either re-create an object through evaluation or obtain a human-readable view of the contents of objects, data values, object types, etc.

```
>>> str(4.53-2j)
'(4.53-2j)'
>>>
>>> str(1)
'1'
>>>>> str(2e10)
```

```
'20000000000.0'  
>>>  
>>> str([0, 5, 9, 9])  
'[0, 5, 9, 9]'  
>>>  
>>> repr([0, 5, 9, 9])  
'[0, 5, 9, 9]'  
>>>  
>>> `[0, 5, 9, 9]`  
'[0, 5, 9, 9]'
```

iv) type() and isinstance()

Python does not support method or function overloading, so you are responsible for any "introspection" of the objects that your functions are called with. Fortunately, we have the type() BIF to help us with just that. Python provides a BIF just for that very purpose. type() returns the type for any Python object, not just the standard types. Using the interactive interpreter, let us take a look at some examples of what type() returns when we give it various objects.

G) Type Factory Functions:

The following familiar factory functions were formerly built-in functions:

- int(), long(), float(), complex()
- str(), unicode(), basestring()
- list(), tuple()
- type()

Other types that did not have factory functions now do. In addition, factory functions have been added for completely new types that support the new-style classes. The following is a list of both types of factory functions:

- dict()
- bool()
- set(), frozenset()
- object()
- classmethod()
- staticmethod()
- super()
- property()
- file()
-

H) Categorizing the Standard Types:

If we were to be maximally verbose in describing the standard types, we would probably call them something like Python's "basic built-in data object primitive types."

- "Basic," indicating that these are the standard or core types that Python provides
- "Built-in," due to the fact that these types come by default in Python
- "Data," because they are used for general data storage
- "Object," because objects are the default abstraction for data and functionality
- "Primitive," because these types provide the lowest-level granularity of data storage
- "Types," because that's what they are: data types!
- However, this description does not really give you an idea of how each type works

i) Storage Model: Types Categorized by the Storage Model

Storage Model Category

Scalar/atom
Container

Python Types That Fit Category

Numbers (all numeric types), strings (all are literals)
Lists, tuples, dictionaries

ii) Update Model: Types Categorized by the Update Model

Update Model Category

Mutable
Immutable

Python Types That Fit Category

Lists, dictionaries
Numbers, strings, tuples

iii) Access Model: Types Categorized by the Access Model

Access Model Category

Direct
Sequence
Mapping

Types That Fit Category

Numbers
Strings, lists, tuples
Dictionaries

Direct types indicate single-element, non-container types.

iv) Categorizing the Standard Types

Data Type	Storage Model	Update Model	Access Model
Numbers	Scalar	Immutable	Direct
Strings	Scalar	Immutable	Sequence
Lists	Container	Mutable	Sequence
Tuples	Container	Immutable	Sequence
Dictionaries	Container	Mutable	Mapping

I) Unsupported Types

- **char or byte:** Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.
- **Pointer:** Since Python manages memory for you, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the `id()` BIF. Since you have no control over this value, it's a moot point.
- **int versus short versus long:** Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, e.g., C's `int`, `short`, and `long`.
- **float versus double:** C has both a single precision float type and double-precision double type. Python's float type is actually a C double. Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types.

3) Explain about numbers in python

- Introduction to Numbers
- Integers
 - a) Boolean
 - b) Standard Integers
 - c) Long Integers
- C. Floating Point Real Numbers
- E. Complex Numbers
- F. Operators
- G. Built-in Functions
- H. Related Modules

A) Introduction to Numbers:

Python has several numeric types: "plain" integers, long integers, Boolean, double-precision floating point real numbers, decimal floating point numbers, and complex numbers. How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable:

```
anInt = 1
```

```
aLong = -9999999999999999L
```

```
aFloat = 3.1415926535897932384626433832795
```

```
aComplex = 1.23+4.56J
```

How to Update Numbers:

You can "update" an existing number by (re)assigning a variable to another number. The new value can

be related to its previous value or to a completely different number altogether.


```
anInt += 1
aFloat = 2.718281828
```

How to Remove Numbers:

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the del statement

```
del anInt
del aLong, aFloat, aComplex
```

B) Integers:

Python has several types of integers. There is the Boolean type with two possible values. There are the regular or plain integers: generic vanilla integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C longs. We will take a look at these types of integers, followed by a description of operators and built-in functions applicable only to Python integer types.

a) **Boolean:** Objects of this type have two possible values, Boolean True and False .

b) Standard (Regular or Plain) Integers:

```
0101 84 -237 0x80 017 -680 -0X92
```

c) **Long Integers:** The first thing we need to say about Python long integers (or long s for short) is not to get them confused with longs in C or other compiled languages these values are typically restricted to 32- or 64- bit sizes, whereas Python longs are limited only by the amount of (virtual) memory in your machine. In other words, they can be very L-O-N-G longs.

d) Unification of Integers and Long Integers:

Both integer types are in the process of being unified into a single integer type.

C) Double Precision Floating Point Numbers:

Floats in Python are implemented as C double s, double precision floating point real numbers, values that can be represented in straightforward decimal or scientific notations.

D) Complex Numbers:

Here are some facts about Python's support of complex numbers:

- Imaginary numbers by themselves are not supported in Python (they are paired with a real part of 0.0 to make a complex number)
- Complex numbers are made up of real and imaginary parts

- Syntax for a complex number: real+imagj
- Both real and imaginary components are floating point values
- Imaginary part is suffixed with letter "J" lowercase (j) or uppercase (J)

Complex Number Attributes

Attribute	description
num.real	Real component of complex number num
num.imag	Imaginary component of complex number num
num.conjugate()	Returns complex conjugate of num

E) Operators:

i) Mixed-Mode Operations:

- If either argument is a complex number, the other is converted to complex;
- Otherwise, if either argument is a floating point number, the other is converted to floating point;
- Otherwise, if either argument is a long, the other is converted to long;
- Otherwise, both must be plain integers and no conversion is necessary (in the upcoming diagram, this describes the rightmost arrow).

ii) Standard Type Operators:

Mixed-mode operations, described above, are those which involve two numbers of different types. The values are internally converted to the same type before the operation is applied. Here are some examples of the standard type operators in action with numbers:

```
>>> 5.2 == 5.2
```

```
True
```

```
>>> -719 >= 833
```

```
False
```

```
>>> 5+4e >= 2-3e
```

```
True
```

```
>>> 2 < 5 < 9
```

iii) Numeric Type (Arithmetic) Operators:

Python supports unary operators for no change and negation, + and - , respectively; and binary arithmetic operators + , - , * , / , % , and ** , for addition, subtraction, multiplication, division, modulo, and exponentiation, respectively. In addition, there is a new division operator, //.

- **Division:**

```
>>> 1 / 2
0
>>> 1.0 / 2.0
0.5
```

- **True Division:**

```
>>> from __future__ import division
>>> 1 / 2 # returns real quotient
0.5
>>> 1.0 / 2.0
0.5
```

Floor Division: A new division operator (//) has been created that carries out floor division.

```
>> 1 // 2 # floors result, returns integer
0
>>> 1.0 // 2.0 # floors result, returns float
0.0
>>> -1 // 2
-1
```

- **Modulus:** Integer modulo is straightforward integer division remainder, while for float, it is the difference of the dividend and the product of the divisor and the quotient of the quantity dividend divided by the divisor rounded down to the closest integer, i.e., $x - (\text{math.floor}(x/y) * y)$, or For complex number modulo, take only the real component of the division result, i.e., $x - (\text{math.floor}((x/y).\text{real}) * y)$.

- **Exponentiation:**

The exponentiation operator has a peculiar precedence rule in its relationship with the unary operators:

```
>>> 3 ** 2
```

9

```
>>> -3 ** 2
```

-9

```
>>> (-3) ** 2
```

9

```
>>> 4.0 ** -1.0
```

0.25

Numeric Type Arithmetic Operators

Arithmetic Operator

Function

`expr1 ** expr2`

expr1 raised to the power of expr2

- `+expr`

(unary) expr sign unchanged

- `-expr`

(unary) negation of expr

- `expr1 ** expr2`

expr1 raised to the power of expr2

- `expr1 * expr2`

expr1 times expr2

- `expr1 / expr2`

expr1 divided by expr2 (classic or true division)

- `expr1 // expr2`

expr1 divided by expr2 (floor division [only])

- `expr1 % expr2`

expr1 modulo expr2

- `expr1 + expr2`

expr1 plus expr2

- `expr1 - expr2`

expr1 minus expr2

*Bit Operators (Integer-Only)

- Python integers may be manipulated bitwise and the standard bit operations are supported: inversion, bitwise AND, OR, and exclusive OR (aka XOR), and left and right shifting. Here are some facts regarding the bit operators:

Negative numbers are treated as their 2's complement value.

- Left and right shifts of N bits are equivalent to multiplication and division by $(2 ** N)$ without overflow checking.

- For longs, the bit operators use a "modified" form of 2's complement, acting as if the sign bitwere extended infinitely to the left.

- The bit inversion operator (`~`) has the same precedence as the arithmetic unary operators, the highest of all bit operators. The bit shift operators (`<<` and `>>`) come next, having a precedence one level below that of the standard plus and minus operators, and finally we have the bitwise AND, XOR, and OR operators (`&` , `^` , `|`), respectively. All of the bitwise operators are presented in the order of descend in priority in

- | Bitwise Operator | Function |
|---------------------------------|--|
| <code>~num</code> | (unary) invert the bits of num , yielding <code>-(num + 1)</code> |
| <code>num1 << num2</code> | num1 left shifted by num2 bits |
| <code>num1 >> num2</code> | num1 right shifted by num2 bits |
| <code>num1 & num2</code> | num1 bitwise AND with num2 |
| <code>num1 ^ num2</code> | num1 bitwise XOR (exclusive OR) with num2 |
| <code>num1 num2</code> | num1 bitwise OR with num2 |
- Here we present some examples using the bit operators using 30 (011110), 45 (101101), and 60 (111100):

F) Built-in and Factory Functions:

i) Standard Type Functions: we introduced the `cmp()` , `str()` , and `type()` built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```
>>> cmp(-6, 2)
```

```
-1
```

```
>>> cmp(-4.333333, -2.718281828)
```

```
-1
```

```
>>> cmp(0xFF, 255)
```

```
0
```

```
>>> str(0xFF)
```

```
'255'
```

```
>>> str(55.3e2)
```

```
'5530.0'
```

```
>>> type(0xFF)
```



```
<type 'int'>
```

```
>>> type(98765432109876543210L)
```

```
<type 'long'>
```

```
>>> type(2-1j)
```

```
<type 'complex'>
```

ii) Numeric Type Functions: Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

- **Conversion Factory Functions:** The `int()`, `long()`, `float()`, and `complex()` functions are used to convert from any numeric type to another.

```
>>> int(4.25555)
```

```
4
```

```
>>> long(42)
```

```
42L
```

```
>>> float(4)
```

```
4.0
```

```
>>> complex(4)
```

```
(4+0j)
```

```
>>>
```

```
>>> complex(2.4, -8)
```

```
(2.4-8j)
```

```
>>>
```

```
>>> complex(2.3e-10, 45.3e4)
```

```
(2.3e-10+453000j)
```

- **Operational:**

Python has five operational built-in functions for numeric types: `abs()`, `coerce()`, `divmod()`, `pow()`, and `round()`. We will take a look at each and present some usage examples. `abs()` returns the absolute value of the given argument. If the argument is a complex number, then `math.sqrt (num .real 2 + num.imag 2)` is returned. Here are some examples of using the `abs()` built-in function:


```
>>> abs(-1)
```

```
1
```

```
>>> abs(10.)
```

```
10.0
```

```
>>> abs(1.2-2.1j)
```

```
2.41867732449
```

```
>>> abs(0.23 - 0.78j)
```

```
0.55
```

- The **coerce()** function: although it technically is a numeric type conversion function, does not convert to a specific type and acts more like an operator, hence our placement of it in our operational built-ins section.

```
>>> coerce(1, 2)
```

```
(1, 2)
```

```
>>>
```

```
>>> coerce(1.3, 134L)
```

```
1.3, 134.0)
```

```
>>>
```

```
>>> coerce(1, 134L)
```

```
(1L, 134L)
```

```
>>>
```

```
>>> coerce(1j, 134L)
```

```
(1j, (134+0j))
```

```
>>>
```

```
>>> coerce(1.23-41j, 134L)
```

```
((1.23-41j), (134+0j))
```

The divmod() built-in function combines division and modulus operations into a single function call that returns the pair (quotient, remainder) as a tuple. The values returned are the same as those given for the classic division and modulus operators for integer types. For floats, the quotient returned is `math.floor (num1/num2)` and for complex numbers, the quotient is `math.floor ((num1/num2). real)`.

```
divmod(10,3)
```

```
(3,1)
```

```
divmod(3,10)
```

```
(0,3)
```

Both pow() and the double star (**) operator perform exponentiation; however, there are differences other than the fact that one is an operator and the other is a built-in function. pow (x , y , z).

```
>>> pow(2,5)
```

```
32
```

```
>>>s
```

```
>>> pow(5,2)
```

```
25
```

```
>>> pow(3.141592,2)
```

```
9.86960029446
```

```
>>>
```

```
>>> pow(1+1j, 3)
```

```
(-2+2j)
```

The round() built-in function: has a syntax of round (flt,ndig =0). It normally rounds a floating point number to the nearest integral number and returns that result (still) as a float. When the optional ndig option is given, round() will round the argument to the specific number of decimal places.

```
>>> round(3)
```

```
3.0
```

```
>>> round(3.45)
```

```
3.0
```

```
>>> round(3.4999999)
```

```
3.0
```

```
>>> round(3.4999999, 1)
```

3.5

```
>>> import math
>>> for eachNum in range(10):
...
print round(math.pi, eachNum)
```

iii) Integer-Only Functions:

- **Base Representation:**

```
>>> hex(255)
'0xff'
>>> hex(230948231)
'0x1606627L'
>>> hex(65535*2)
'0x1fffe'
>>> oct(255)
'0377'
>>> oct(230948231)
'0130063047L'
>>> oct(65535*2)
'0377776'
```

iv) ASCII Conversion:

Python also provides functions to go back and forth between ASCII (American Standard Code for

Information Interchange) characters and their ordinal integer values.

```
>>> ord('a')
97
>>> ord('A')
65
```

```
>>> ord('0')
48
>>> chr(97)
'a'
>>> chr(65L)
'A'
>>> chr(48)
'0'
```

G) other Built-in Functions

Function	Operation
hex(num)	Converts num to hexadecimal and returns as string
oct(num)	Converts num to octal and returns as string
chr(num)	Takes ASCII value num and returns ASCII character as string; $0 \leq \text{num} \leq 255$ only
ord(chr)	Takes ASCII or Unicode chr (string of length 1) and returns corresponding ASCII value or Unicode code point, respectively
unichr(num)	Takes a Unicode code point value num and returns its Unicode character as a Unicode string; valid range depends on whether your Python was built as UCS-2 or UCS-4

H) Related Modules

here are a number of modules in the Python standard library that add on to the functionality of the operators and built-in functions for numeric types.

Numeric Type Related Modules

Module	Contents
decimal --	Decimal floating point class Decimal
array --	Efficient arrays of numeric values (characters, ints, floats, etc.)
math/cmath --	Standard C library mathematical functions; most functions available in math are implemented for complex numbers in the cmath module.
Operator --	equivalent to the difference (m - n) for numbers m and n

random -- Various pseudo-random number generators (obsoletes rand and
 mathematics wHRandom) For advanced numerical and scientific applications, there are well-k

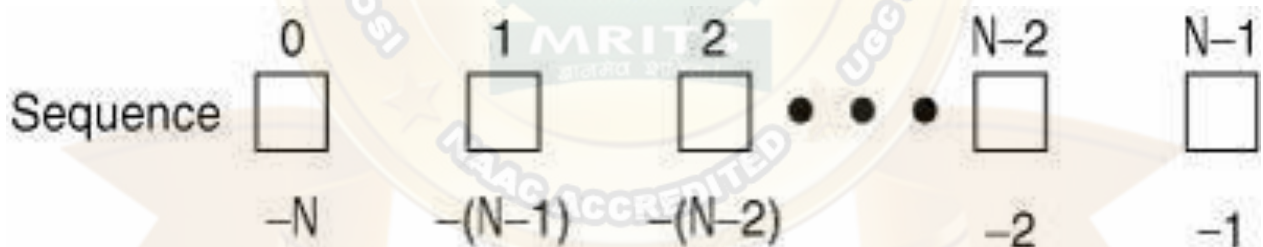
4) Explain about Sequences: Strings, Lists, and Tuples

- Introduction to Sequences
- Strings
- Lists
- Tuples

A) Introduction to Sequences:

Sequence types all share the same access model: ordered set with sequentially indexed offsets to get to each element. Multiple elements may be selected by using the slice operators, which we will explore in this chapter.

How sequence elements are stored and accessed



$N == \text{length of sequence} == \text{len}(\text{sequence})$

i) Standard Type Operators: A list of all the operators applicable to all sequence types is given in Table .The operators appear in hierarchical order from highest to lowest with the levels alternating between shaded and not.

Sequence Type Operators

Sequence Operator	Function
seq[ind]	Element located at index ind of seq
seq[ind1 : ind2]	Elements from ind1 up to but not including ind2 of seq
seq * expr	seq repeated expr times
seq1 + seq2	Concatenates sequences seq1 and seq2

obj in seq Tests if obj is a member of sequence seq

obj not in seq Tests if obj is not a member of sequence seq

Membership (in, not in):

Membership test operators are used to determine whether an element is in or is a member of a sequence. For strings, this test is whether a character is in a string, and for lists and tuples, it is whether an object is an element of those sequences. The in and not in operators are Boolean in nature; they return true if the membership is confirmed and False otherwise.

The syntax for using the membership operators is as follows:

obj [not] in sequence

Concatenation (+): This operation allows us to take one sequence and join it with another sequence of the same type. The syntax for using the concatenation operator is as follows:

sequence1 + sequence2

The resulting expression is a new sequence that contains the combined contents of sequence1 and sequence2.

Repetition (*)

The repetition operator is useful when consecutive copies of sequence elements are desired. The syntax for using the repetition operator is as follows:

sequence * copies_int

The number of copies, copies_int, must be an integer (prior to 1.6, long integers were not allowed). As with the concatenation operator, the object returned is newly allocated to hold the contents of the multiply replicated objects.

Slices ([], [:], [: :])

To put it simply: sequences are data structures that hold objects in an ordered manner. You can get access to individual elements with an index and pair of brackets, or a consecutive group of elements with the brackets and colons giving the indices of the elements you want starting from one index and going up to but not including the ending index.

The **syntax** for accessing an individual element is:

sequence[index]

sequence is the name of the sequence and index is the offset into the sequence where the desired element is located.

B) Explain about string

- Strings:

- string and operator
- string only operator
- built- in - function
- string Built -in-methods
- special features of string
- Related modules

a)Strings:

Strings are among the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other shell-type scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters.

- **How to Create and Assign Strings**

Creating strings is as simple as using a scalar value or having the str() factory function make one and assigning it to a variable:Creating strings is as simple as using a scalar value or having the str() factory function make one and assigning it to a variable:

```
>>> aString = 'Hello World!'          # using single quotes
```

```
>>> anotherString = "Python is cool!" # double quotes
```

```
>>> print aString                     # print, no quotes!
```

```
Hello World!
```

```
>>> anotherString                    # no print, quotes!
```

```
'Python is cool!'
```

```
>>> s = str(range(4))                # turn list to string
```

```
>>> s
```

```
'[0, 1, 2, 3]'
```

- **How to Access Values (Characters and Substrings) in Strings**

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
```

```
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

- **How to Update Strings**

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

- **How to Remove Characters and Strings**

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string that drops the pieces you were not interested in.

Let us say you want to remove one letter from "Hello World!"...the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

- **To clear or remove a string, you assign an empty string or use the del statement, respectively:**

```
>>> aString = ''
```

```
>>> aString
```

```
"
```

```
>>> del aString
```

b) Strings and Operators

a) Standard Type Operators

a number of operators that apply to most objects, including the standard

types. We will take a look at how some of those apply to strings. For a brief introduction, here are a few

examples using strings:

```
>>> str1
```

b) Sequence Operators

-Slices ([] and [:])

Earlier in Section 6.1.1, we examined how we can access individual or a group of elements from a

sequence. We will apply that knowledge to strings in this section. In particular, we will look at:

- Counting forward
- Counting backward
- Default/missing indexes

For the following examples, we use the single string 'abcd' . Provided in the figure is a list of positive

and negative indexes that indicate the position in which each character is located within the string itself.

Using the length operator, we can confirm that its length is 4:

```
>>> aString = 'abcd'
```

```
>>> len(aString)
```

-Membership (in, not in):

The membership question asks whether a (sub)string appears in a (nother) string. true is returned if

that character appears in the string and False otherwise.

```
>>> 'bc' in 'abcd'
```

True

```
>>> 'n' in 'abcd'
```

False

```
>>> 'nm' not in 'abcd'
```

True

predefined strings found in the string module:

```
>>> import string
```

```
>>> string.uppercase
```

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
>>> string.lowercase
```

```
'abcdefghijklmnopqrstuvwxyz'
```

```
>>> string.letters
```

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
>>> string.digits
```

```
'0123456789'
```

c)String-Only Operators

a)Format Operator (%)

Python features a string format operator. This operator is unique to strings and makes up for the lack of having functions from C's printf() family. In fact, it even uses the same symbol, the percent sign (%), and supports all the printf() formatting codes.

The syntax for using the format operator is as follows:

```
format_string % (arguments_to_convert)
```

The format_string on the left-hand side is what you would typically find as the first argument to printf() : the format string with any of the embedded % codes. The set of valid codes is given in Table 6.4. The arguments_to_convert parameter matches the remaining arguments you would send to printf() , namely the set of variables to convert and display.

d)Built-in Functions:

a)Standard Type Functions

-cmp()

As with the value comparison operators, the `cmp()` built-in function also performs a lexicographic (ASCII value-based) comparison for strings.

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> cmp(str1, str2)
-11
>>> cmp(str3, str1)
23
>>> cmp(str2, 'lmn')
0
```

b) Sequence Type Functions

- **len()**

```
>>> str1 = 'abc'
>>> len(str1)
3
>>> len('Hello World!')
12
```

The len() built-in function returns the number of characters in the string as expected.

- **max() and min()**

```
str2 = 'lmn'
str3 = 'xyz'
max(str2)
n
min(str3)
x
```

- **enumerate()**

```
>>> s = 'foobar'
>>> for i, t in enumerate(s):
... print i, t
...
```

- **zip()**

```
>>> s, t = 'foa', 'obr'
>>> zip(s, t)
[(f, 'o'), (o, 'b'), (a, 'r')]
```

c) String Type Functions

- **raw_input():**

The built-in `raw_input()` function prompts the user with a given string and accepts and returns a user-

input string. Here is an example using `raw_input()` :

```
>>> user_input = raw_input("Enter your name: ")
```

```
Enter your name: John Doe
```

```
>>>
```

```
>>> user_input
```

```
'John Doe'
```

```
>>>
```

```
>>> len(user_input)
```

```
8
```

- **str() and unicode()**

Both `str()` and `unicode()` are factory functions, meaning that they produce new objects of their type respectively. They will take any object and create a printable or Unicode string representation of the argument object. And, along with `basestring`, they can also be used as arguments along with objects `isinstance()` calls to verify type:

```
>>> isinstance(u'\0xAB', str)
```

```
False
```

```
>>> not isinstance('foo', unicode)
```

```
True
```

```
>>> isinstance(u'', basestring)
```

```
True
```

```
>>> not isinstance('foo', basestring)
```

```
False
```

- **chr() , unichr() , and ord()**

chr() takes a single integer argument in range(256) (e.g., between 0 and 255) and returns the corresponding character

```
>>> chr(65)
```

```
'A'
```

```
>>> ord('a')
```

```
9
```

e)String Built-in Methods

String Type Built-in Methods

Method Name	Description
string.capitalize()	Capitalizes first letter of string
string.center(width)	Returns a space-padded string with the original string centered to a total of width columns
string.count(str, beg= 0, end=len(string))	Counts how many times str occurs in string , or in a substring of string if starting index beg and ending index end are given

g)Related Modules:

Related Modules for String Types

Module	Description
string	String manipulation and utility functions, i.e., Template class
re	Regular expressions: powerful string pattern matching

struct	Convert strings to/from binary data format
c/StringIO	String buffer object that behaves like a file
base64	Base 16, 32, and 64 data encoding and decoding
codecs	Codec registry and base classes
crypt	Performs one-way encryption cipher

5) EXPLAIN ABOUT LIST

- list
- operator
- Built-in-Function
- List type Built-in- methods
- special feature of list

a. List

Like strings, lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there.

- **How to Create and Assign Lists**

Creating lists is as simple as assigning a value to a variable. You handcraft a list (empty or with elements) and perform the assignment. Lists are delimited by surrounding square brackets ([]). You

can also use the factory function.

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
```

```
>>> anotherList = [None, 'something to see here']
```

```
>>> print aList
```

```
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
```

```
>>> print anotherList
```

```
[None, 'something to see here']
```

```
>>> aListThatStartedEmpty = []
```

```
>>> print aListThatStartedEmpty
```

```
[]
```

```
>>> list('foo')
```

```
['f', 'o', 'o']
```

- **How to Access Values in Lists**

Slicing works similar to strings; use the square bracket slice operator ([]) along with the index or

indices.

```
>>> aList[0]
```

```
123
```

```
>>> aList[1:4]
```

```
['abc', 4.56, ['inner', 'list']]
```

```
>>> aList[:3]
```

```
[123, 'abc', 4.56]
```

```
>>> aList[3][1]
```

```
'list'
```

- **How to Update Lists**

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method:

```
>>> aList
```

```
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
```

```
>>> aList[2]
```

```
4.56
```

```
>>> aList[2] = 'float replacer'
```

```
>>> aList
```

```
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
```

```
>>>
```

```
>>> anotherList.append("hi, i'm new here")
```



```
>>> print anotherList
[None, 'something to see here', 'hi, i'm new here']
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty
['not empty anymore']
```

- **How to Remove List Elements and Lists**

To remove a list element, you can use either the del statement if you know exactly which element(s)

you are deleting or the remove() method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

b)Operators

a)Standard Type Operators

```
>>>list1 = ['abc',12]
>>>list2 =['ABC',126]
list1 < list2
true
```

b) Sequence Type Operators

Slices ([] and [:])

Slicing with lists is very similar to strings, but rather than using individual characters or substrings, slices of lists pull out an object or a group of objects that are elements of the list operated on. Focusing specifically on lists, we make the following definitions:

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

Membership (in , not in)

With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
True
>>>
>>> 'x' in mixup_list
False
>>>
>>> 'x' in mixup_list[1]
True
>>> num_list
[[65535L, 2e+030, (76.45-1.3j)], -1.23, 16.0, -49]
>>>
>>> -49 in num_list
True
>>>
>>> 34 in num_list
False
```

```
>>>
```

```
>>> [65535L, 2e+030, (76.45-1.3j)] in num_list
```

```
true
```

Concatenation (+):

The concatenation operator allows us to join multiple lists together.

Repetition (*)

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and

tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
```

```
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
```

```
>>>
```

```
>>> num_list * 3
```

```
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43,  
-1.23, -2, 619000.0]
```

```
>>>
```

C) Built-in Functions

i) Standard Type Functions

cmp():

```
>>> list1, list2 = [123, 'xyz'], [456, 'abc']
```

```
>>> cmp(list1, list2)
```

```
-1
```

```
>>>
```

```
>>> cmp(list2, list1)
```

```
1
```

```
>>> list3 = list2 + [789]
```

```
>>> list3
```

```
[456, 'abc', 789]
```

```
>>>
```

```
>>> cmp(list2, list3)
```

ii) Sequence Type Functions

len()

```
>>> len(num_list)
```

```
4
```

```
>>>
```

```
>>> len(num_list*2)
```

```
8
```

max() and min()

```
>>> max(str_list)
```

```
'park'
```

```
>>> max(num_list)
```

```
[65535L, 2e+30, (76.45-1.3j)]
```

```
>>> min(str_list)
```

```
'candlestick'
```

```
>>> min(num_list)
```

```
-49
```

sorted() and reversed()

```
>>> s = ['They', 'stamp', 'them', 'when', "they're", 'small']
```

```
>>> for t in reversed(s):
```

```
... print t,
```

```
...
```

```
small they're when them stamp They
```

```
>>> sorted(s)
```

```
['They', 'small', 'stamp', 'them', "they're", 'when']
```

enumerate() and zip()

```
>>> albums = ['tales', 'robot', 'pyramid']
```

```
>>> for i, album in enumerate(albums):
```

```
...
```

```
print i, album
```

```
...
```

```
0 tales
```

```
1 robot
```

```
2 pyramid
```

```
>>>
```

```
>>> fn = ['ian', 'stuart', 'david']
```

```
>>> ln = ['bairnson', 'elliott', 'paton']
```

```
>>>
```

```
>>> for i, j in zip(fn, ln):
```

```
...
```

```
print ('%s %s' % (i,j)).title()
```

```
...
```

```
Ian Bairnson
```

```
Stuart Elliott
```

```
David Paton
```

```
sum()
```

```
>>> a = [6, 4, 5]
```

```
>>> reduce(operator.add, a)
```

```
15
```

```
>>> sum(a)
```

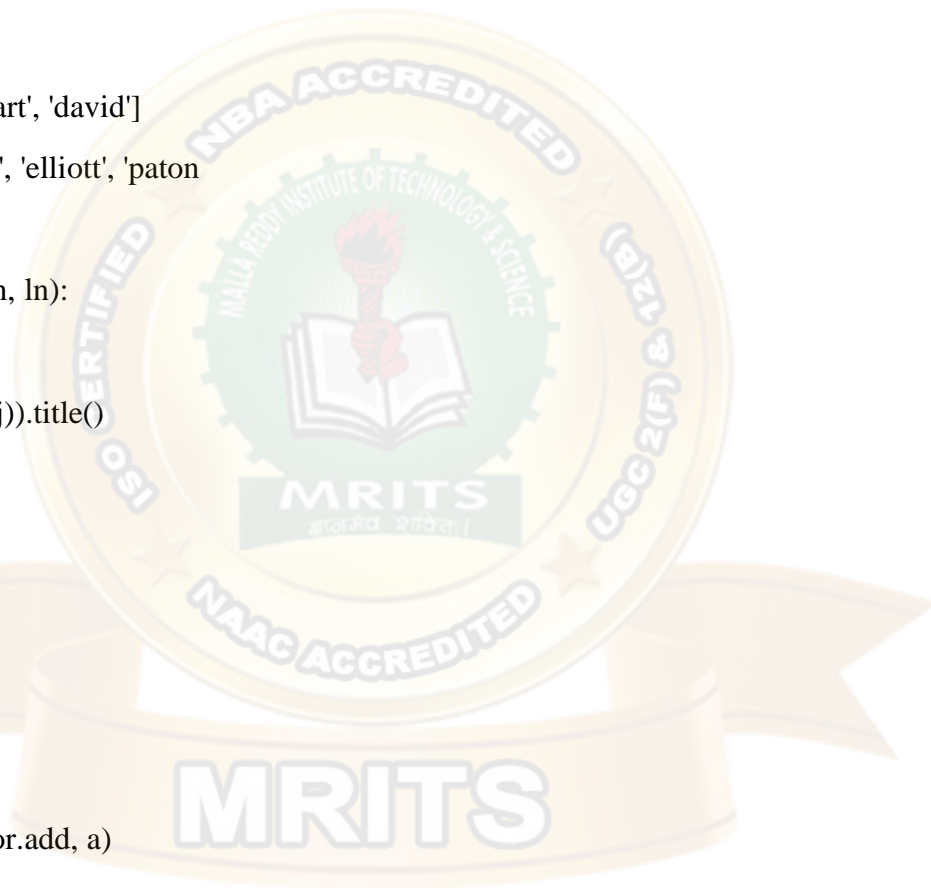
```
15>>> sum(a, 5)
```

```
20
```

```
>>> a = [6., 4., 5.]
```

```
>>> sum(a)
```

```
15.0
```



list() and tuple()

The list() and tuple() factory functions take iterables like other sequences and make new lists and tuples, respectively, out of the (just shallow-copied) data. Although strings are also sequence types,

they are not commonly used with list() and tuple() . These built-in functions are used more often to

convert from one type to the other, i.e., when you have a tuple that you need to make a list (so that

you can modify its elements) and vice versa.

```
>>> aList = ['tao', 93, 99, 'time']
```

```
>>> aTuple = tuple(aList)
```

```
>>> aList, aTuple
```

```
(['tao', 93, 99, 'time'], ('tao', 93, 99, 'time'))
```

```
False
```

```
>>> anotherList = list(aTuple)
```

```
>>> aList == anotherList
```

```
True
```

```
>>> aList is anotherList
```

```
False
```

```
>>> [id(x) for x in aList, aTuple, anotherList]
```

```
[10903800, 11794448, 11721544]
```

c) List Type Built-in Functions

There are currently no special list-only built-in functions in Python unless you consider range() as one its sole function is to take numeric input and generate a list that matches the criteria. range() is covered in Lists can be used with most object and sequence built-in functions. In addition, list objects have their own methods.

d) List Type Built-in Methods

List Method	Operation
list.append(obj)	Adds obj to the end of list
list.count(obj)	Returns count of how many times obj occurs in List

e) Special Features of Lists

a) Creating Other Data Structures Using Lists

Because of their container and mutable features, lists are fairly flexible and it is not very difficult to build other kinds of data structures using lists. Two that we can come up with rather quickly are stacks and queues.

-Stack

A stack is a last-in-first-out (LIFO) data structure that works similarly to a cafeteria dining plate spring-loading mechanism. Consider the plates as objects. The first object off the stack is the last one you put in. Every new object gets "stacked" on top of the newest objects. To "push" an item on a stack is the terminology used to mean you are adding onto a stack

6) Explain about tuples

- Tuple
- Tuple Operator and Built in Functions
- Special Features of Tuples
- Related Modules
- *copying Python Objects and shallow and deep copies

a) Tuple

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally, there is a more significant difference, and that is the fact that tuples are immutable. Because of this, tuples can do something that lists cannot do . . . be a dictionary key. Tuples are also the default when dealing with a group of objects.

How to Create and Assign Tuples

Creating and assigning tuples are practically identical to creating and assigning lists, with the exception of empty tuples these require a trailing comma (,) enclosed in the tuple delimiting

parentheses (()) to prevent them from being confused with the natural grouping operation of parentheses. Do not forget the factory function!

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
```

```
>>> anotherTuple = (None, 'something to see here')
```

```
>>> print aTuple
```

```
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
```

```
>>> print anotherTuple
```

```
(None, 'something to see here')
```

```
>>> emptiestPossibleTuple = (None,)
```

```
>>> print emptiestPossibleTuple
```

```
(None,)
```

```
>>> tuple('bar')
```

```
('b', 'a', 'r')
```

How to Access Values in Tuples

Slicing works similarly to lists. Use the square bracket slice operator ([]) along with the index or

indices.

```
>>> aTuple[1:4]
```

```
('abc', 4.56, ['inner', 'tuple'])
```

```
>>> aTuple[:3]
```

```
(123, 'abc', 4.56)
```

```
>>> aTuple[3][1]
```

```
'tuple'
```

How to Update Tuples

Like numbers and strings, tuples are immutable, which means you cannot update them or change values of tuple elements.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
```

```
>>> aTuple
```

```
(123, 'abc', (7-9j))
```

```
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To explicitly remove an entire tuple, just use the del statement to reduce an object's reference count. It will be deallocated when that count is zero. Keep in mind that most of the time one will just let an object go out of scope rather than using del, a rare occurrence in everyday Python programming.

Del a Tuple

b) Tuple Operators and Built-in Functions

- **Standard and Sequence Type Operators and Built-in Functions**

Object and sequence operators and built-in functions act the exact same way toward tuples as they do

with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate

membership, and compare tuples.

Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

Membership, Slicing

```
>>> 23 in t
```

```
True
```

```
>>> 123 in t
```

```
False
```

```
>>> t[0][1]
```

```
123
```

```
>>> t[1:]
```

```
(23, -103.4, 'free', 'easy')
```

Built-in Functions

```
>>> str(t)
```

```
('['xyz', 123], 23, -103.4, 'free', 'easy')
```

```
>>> len(t)
```

```
5
```

```
>>> max(t)
```

```
'free'
```

```
>>> min(t)
```

```
-103.4
```

```
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free', 'easy'))
```

```
0
```

```
>>> list(t)
```

```
['xyz', 123], 23, -103.4, 'free', 'easy']
```

Operators

```
>>> (4, 2) < (3, 5)
```

```
False
```

```
>>> (2, 4) < (3, -1)
```

```
True
```

```
>>> (2, 4) == (3, -1)
```

```
False
```



```
>>> (2, 4) == (2, 4)
```

```
True
```

ii) Tuple Type Operators and Built-in Functions and Methods

Like lists, tuples have no operators or built-in functions for themselves. All of the list methods described in the previous section were related to a list object's mutability, i.e., sorting, replacing, appending, etc. Since tuples are immutable, those methods are rendered superfluous, thus unimplemented.

c) Special Features of Tuples

a) How Are Tuples Affected by Immutability?

Okay, we have been throwing around this word "immutable" in many parts of the text. Aside from its

computer science definition and implications, what is the bottom line as far as applications are concerned? What are all the consequences of an immutable data type? Of the three standard types that are immutable—numbers, strings, and tuples—tuples are the most affected. A data type that is immutable simply means that once an object is defined, its value cannot be updated, unless, of course, a completely new object is allocated. The impact on numbers and strings is not as great since they are scalar types, and when the sole value they represent is changed, that is the intended effect, and access occurs as desired. The story is different with tuples, however.

b) Tuples Are Not Quite So "Immutable"

Although tuples are defined as immutable, this does not take away from their flexibility. Tuples are not quite as immutable as we made them out to be. What do we mean by that? Tuples have certain behavioral characteristics that make them seem not as immutable as we had first advertised.

```
>>> s = 'first'
```

```
>>> s = s + ' second'
```

```
>>> s
```

```
'first second'
```

```
>>>
```

```
>>> t = ('third', 'fourth')
```

```
>>> t
```

```
('third', 'fourth')
```

```
>>>
```

```
>>> t = t + ('fifth', 'sixth')
```

```
>>> t
```

```
('third', 'fourth', 'fifth', 'sixth')
```

c) Default Collection Type

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., defaults to tuples, as indicated in these short examples:

```
>>> 'abc', -4.24e93, 18+6.6j, 'xyz'
```

```
('abc', -4.24e+093, (18+6.6j), 'xyz')
```

```
>>>
```

```
>>> x, y = 1, 2
```

```
>>> x, y
```

```
(1, 2)
```

Any function returning multiple objects (also no enclosing symbols) is a tuple. Note that enclosing symbols change a set of multiple objects returned to a single container object. For example:

```
def foo1():
```

```
:
```

```
return obj1, obj2, obj3
```

```
def foo2():
```

```
:
```

```
return [obj1, obj2, obj3]
```

```
def foo3():
```

```
:
```

```
return (obj1, obj2, obj3)
```

d) Single-Element Tuples

Ever try to create a tuple with a single element? You tried it with lists, and it worked, but then you tried and tried with tuples, but you cannot seem to do it.

```
>>> ['abc']
```

```

['abc']
>>> type(['abc'])
<type 'list'>
>>>
>>> ('xyz')
'xyz'
>>> type(('xyz'))
<type 'str'>

```

e) Dictionary Keys

Immutable objects have values that cannot be changed. That means that they will always hash to the same value. That is the requirement for an object being a valid dictionary key. As we will find out in the next chapter, keys must be hashable objects, and tuples meet that criteria. Lists are not eligible.

d) Related Modules

lists the key related modules for sequence types.

Module	Contents
array all of	Features the array restricted mutable sequence type, which requires its elements to be of the same type
copy objects	Provides functionality to perform shallow and deep copies of

g)* Copying Python Objects and Shallow and Deep Copies

we described how object assignments are simply object references. This means that when you create an object, then assign that object to another variable, Python does not copy the object. Instead, it copies only a reference to the object.

```

>> person = ['name', ['savings', 100.00]]
>>> hubby = person[:]
# slice copy
>>> wifey = list(person)
# fac func copy
>>> [id(x) for x in person, hubby, wifey]

```

[11826320, 12223552, 11850936]

7) Explain about Mapping and Set Types

1) Mapping Type: Dictionaries

- Operators
- Built-in Functions
- Built-in Methods
- Dictionary Keys

2. Set Types

- Operators
- Built-in Functions
- Built-in Methods

3. Related Modules

a) explain about Mapping Type: Dictionaries

- Operators
- Built-in Functions
- Built-in Methods
- Dictionary Keys

Dictionaries are the sole mapping type in Python. Mapping objects have a one-to-many correspondence between hashable values (keys) and the objects they represent (values). They are similar to Perl hashes and can be generally considered as mutable hash tables.

How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}  
>>> dict2 = {'name': 'earth', 'port': 80}  
>>> dict1, dict2  
({}, {'port': 80, 'name': 'earth'})
```

How to Access Values in Dictionaries

To traverse a dictionary (normally by key), you only need to cycle through its keys, like this:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2.keys():
...
print 'key=%s, value=%s' % (key, dict2[key])
...>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2.keys():
...
print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth
key=port, value=80
key=name, value=earth
key=port, value=80
```

How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry (see below for more details on removing an entry).

```
>>> dict2['name'] = 'venus' # update existing entry
>>> dict2['port'] = 6969
# update existing entry>>> dict2['arch'] = 'sunos5' # add new entry
>>>
>>> print 'host %(name)s is running on port %(port)d' %
dict2
host venus is running on port 6969
```

How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary. However, if you really want to "remove" an entire dictionary, use the del statement (introduced in Section 3.5.5). Here are some deletion

examples for dictionaries and dictionary elements:

```
del dict2['name']
```

```
dict2.clear()
```

```
del dict2
```

```
dict2.pop('name')
```

b) explain about Mapping Type Operators

i) Standard Type Operators

The standard type operators were introduced in Chapter 4. Here are some basic examples using some of

those operators:

```
>>> dict4 = {'abc':123}
```

```
>>> dict5 = {'abc':456}
```

```
>>> dict6 = {'abc':123, 98.6: 37}
```

```
>>> dict7 = {'xyz':123}
```

```
True
```

```
>>> (dict4 < dict6) and (dict4 < dict7)
```

```
True
```

```
>>> (dict5 < dict6) and (dict5 < dict7)
```

```
True
```

```
>>> dict6 < dict7
```

```
False
```

ii) Mapping Type Operators

Dictionary Key-Lookup Operator ([])

The only operator specific to dictionaries is the key-lookup operator, which works very similarly to the single element slice operator for sequence types. For sequence types, an index offset is the sole argument or subscript to access a single element of a sequence. For a dictionary, lookups are by key, so that is the argument rather than an index. The key-lookup operator is used for both assigning values to and retrieving values from a dictionary:

```
d[k] = v # set value 'v' in dictionary with key 'k'
```

```
d[k] # lookup value in dictionary with key 'k'
```

c) explain about Mapping Type Built-in and Factory Functions

i) Standard Type Functions [type(), str() , and cmp()]

The type() factory function, when applied to a dict, returns, as you might expect, the dict type, "<type

'dict'> ". The str() factory function will produce a printable string representation of a dictionary.

These

are fairly straightforward.

*Dictionary Comparison Algorithm

In the following example, we create two dictionaries and compare them, then slowly modify the dictionaries to show how these changes affect their comparisons:

```
>>>dict1 = {}
```

```
>>>dict2 = {'host': 'earth', 'port': 80}
```

```
>>>cmp(dict1, dict2)
```

```
-1
```

```
>>>dict1['host'] = 'earth'
```

```
>>>cmp(dict1, dict2)
```

```
-1
```

In the first comparison, dict1 is deemed smaller because dict2 has more elements (2 items vs. 0 items). After adding one element to dict1 , it is still smaller (2 vs. 1), even if the item added is also in

dict2.

```
>>>dict1['port'] = 8080
```

```
>>>cmp(dict1, dict2)
```

1

```
>>>dict1['port'] = 80
```

```
>>>cmp(dict1, dict2)
```

0

Our final example reminds us that `cmp()` may return values other than -1, 0, or 1. The algorithm pursues comparisons in the following order.

(1) Compare Dictionary Sizes

If the dictionary lengths are different, then for `cmp(dict1, dict2)`, `cmp()` will return a positive number if `dict1` is longer and a negative number if `dict2` is longer. In other words, the dictionary with more keys is greater, i.e.,

```
len(dict1) > len(dict2)
```

```
dict1 > dict2
```

(2) Compare Dictionary Keys

If both dictionaries are the same size, then their keys are compared; the order in which the keys are checked is the same order as returned by the `keys()` method. (It is important to note here that keys that are the same will map to the same locations in the hash table. This keeps key-checking consistent.) At the point where keys from both do not match, they are directly compared and `cmp()` will return a positive number if the first differing key for `dict1` is greater than the first differing key of `dict2`.

(3) Compare Dictionary Values

If both dictionary lengths are the same and the keys match exactly, the values for each key in both dictionaries are compared. Once the first key with non-matching values is found, those values are compared directly. Then `cmp()` will return a positive number if, using the same key, the value in `dict1` is greater than the value in `dict2`.

(4) Exact Match If we have reached this point, i.e., the dictionaries have the same length, the same keys, and the same values for each key, then the dictionaries are an exact match and 0 is returned.

ii) Mapping Type Related Functions

`dict()`

The `dict()` factory function is used for creating dictionaries. If no argument is provided, then an empty dictionary is created. The fun happens when a container object is passed in as an argument to `dict()`. If the argument is an iterable, i.e., a sequence, an iterator, or an object that supports iteration, then each element of the iterable must come in pairs. For each pair, the first element

will be a new key in the dictionary with the second item as its value. Taking a cue from the official Python documentation for dict

```
():>>> dict(zip(('x', 'y'), (1, 2)))
{'y': 2, 'x': 1}
>>> dict(['x', 1], ['y', 2])
{'y': 2, 'x': 1}
>>> dict([('xy'[i-1], i) for i in range(1,3)])
{'y': 2, 'x': 1}
```

len()

The len() BIF is flexible. It works with sequences, mapping types, and sets (as we will find out later on in this chapter). For a dictionary, it returns the total number of items, that is, key-value pairs:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict2
{'port': 80, 'name': 'earth'}
>>> len(dict2)
2
```

hash()

The hash() BIF is not really meant to be used for dictionaries per se, but it can be used to determine whether an object is fit to be a dictionary key (or not). Given an object as its argument, hash() returns the hash value of that object.

Mapping Type Related Functions

Function	Operation
dict([container]) from	Factory function for creating a dictionary populated with items container , if provided; if not, an empty dict is created
len(mapping)	Returns the length of mapping (number of key-value pairs)
hash(obj)	Returns hash value of obj

d) explain about Mapping Type Built-in Methods

Dictionaries have an abundance of methods to help you get the job done,

Dictionary Type

Methods

`dict.has_key(key)` Returns True if key is in dict, False otherwise; partially deprecated by the `in` and `not in` operators in 2.2 but still provides a functional interface

explain about Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, i.e., from standard objects to user-defined objects.

i) More Than One Entry per Key Not Allowed

One rule is that you are constrained to having only one entry per key. In other words, multiple values per the same key are not allowed. (Container objects such as lists, tuples, and other dictionaries are fine.) When key collisions are detected (meaning duplicate keys encountered during assignment), the last (most recent) assignment wins.

```
>>> dict1 = {'foo':789, 'foo': 'xyz'}
```

```
>>> dict1
```

```
{'foo': 'xyz'}
```

```
>>>
```

```
>>> dict1['foo'] = 123
```

```
>>> dict1
```

```
{'foo': 123}
```

ii) Keys Must Be Hashable

As we mentioned earlier in Section 7.1, most Python objects can serve as keys; however they have to be hashable objects mutable types such as lists and dictionaries are disallowed because they cannot be hashed. All immutable types are hashable, so they can definitely be used as keys. One caveat is numbers: Numbers of the same value represent the same key. In other words, the integer 1 and the float 1.0 hash to the same value, meaning that they are identical as keys.

Explain about Set Types

In mathematics, a set is any collection of distinct items, and its members are often referred to as set elements. Python captures this essence in its set type objects. A set object is an unordered collection of hashable values. Yes, set members would make great dictionary keys. Mathematical sets translate to Python set objects quite effectively and testing for set membership and operations such as union and intersection work in Python as expected. Like other container types, sets support membership testing via `in` and `not in` operators, cardinality using the `len()` BIF, and

iteration over the set membership using for loops. However, since sets are unordered, you do not index into or slice them, and there are no keys used to access a value. There are two different types of sets available, mutable (set) and immutable (frozenset). As you can imagine, you are allowed to add and remove elements from the mutable form but not the immutable. Note that mutable sets are not hashable and thus cannot be used as either a dictionary key or as an element of another set. The reverse is true for frozen sets, i.e., they have a hash value and can be used as a dictionary key or a member of a set.

How to Create and Assign Set Types

There is no special syntax for sets like there is for lists ([]) and dictionaries ({ }) for example. Lists and dictionaries can also be created with their corresponding factory functions list() and dict() , and that is also the only way sets can be created, using their factory functions set() and frozenset() :

```
>>> s = set('cheeseshop')
```

```
>>> s
```

```
set(['c', 'e', 'h', 'o', 'p', 's'])
```

```
>>> t = frozenset('bookshop')
```

```
>>> t
```

```
frozenset(['b', 'h', 'k', 'o', 'p', 's'])
```

```
>>> type(s)
```

```
<type 'set'>
```

```
>>> type(t)
```

```
<type 'frozenset'>
```

```
>>> len(s)
```

```
6
```

```
>>> len(s) == len(t)
```

```
True
```

```
>>> s == t
```

```
False
```


How to Access Values in Sets

You are either going to iterate through set members or check if an item is a member (or not) of a set:

```
>>> 'k' in s
```

```
False
```

```
>>> 'k' in t
```

```
True
```

```
>>> 'c' not in t
```

```
True
```

```
>>> for i in s:
```

```
... print i
```

```
...
```

```
c
```

```
e
```

```
h
```

```
o
```

```
p
```

```
s
```

How to Update Sets

You can add and remove members to and from a set using various built-in methods and operators:

```
>>> s.add('z')
```

```
>>> s
```

```
set(['c', 'e', 'h', 'o', 'p', 's', 'z'])
```

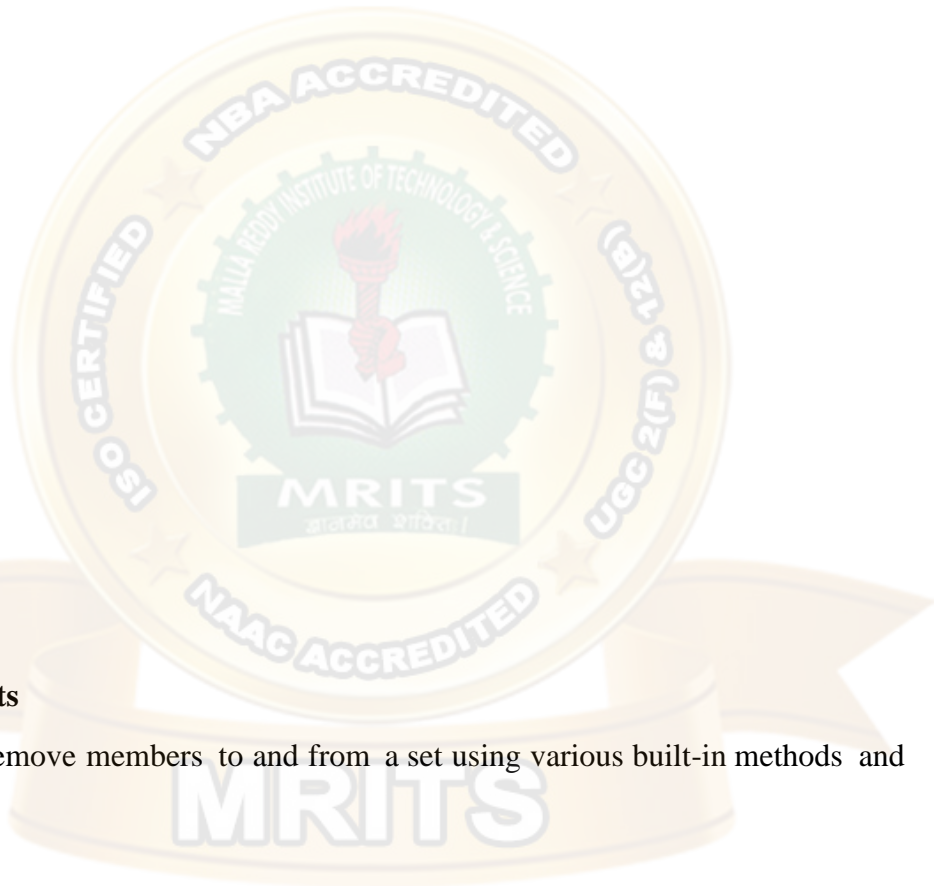
```
>>> s.update('pypi')
```

```
>>> s
```

```
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y', 'z'])
```

```
>>> s.remove('z')
```

```
>>> s
```




```
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
```

```
>>> s -= set('pypi')
```

```
>>> s
```

```
set(['c', 'e', 'h', 'o's'])
```

How to Remove Set Members and Sets

We saw how to remove set members above. As far as removing sets themselves, like any Python object, you can let them go out of scope or explicitly remove them from the current namespace with `del`. If the reference count goes to zero, then it is tagged for garbage collection.

```
>>> del s
```

```
>>>
```

A) Set Type Operators

i) Standard Type Operators (all set types)

- **Membership (in, not in)**

As for sequences, Python's `in` and `not in` operators are used to determine whether an element is (or is

not) a member of a set.

```
>>> s = set('cheeseshop')
```

```
>>> t = frozenset('bookshop')
```

```
>>> 'k' in s
```

```
False
```

```
>>> 'k' in t
```

```
True
```

```
>>> 'c' not in t
```

```
True
```

Set Equality/Inequality

Equality (or inequality) may be checked between the same or different set types. Two sets are equal if and only if every member of each set is a member of the other. You can also say that each set must be a (n improper) subset of the other, e.g., both expressions `s <= t` and `s >= t` are true, or `(s <= t and s >= t)` is `TRue`. Equality (or inequality) is independent of set type or ordering of members when the sets were created it is all based on the set membership.

```
>>> s == t
```

```
False
```

```
>>> s != t
```

```
True
```

```
>>> u = frozenset(s)
```

```
>>> s == u
```

```
True
```

```
>>> set('posh') == set('shop')
```

```
True
```

Subset Of/Superset Of

Sets use the Python comparison operators to check whether sets are subsets or supersets of other sets. The "less than" symbols (<, <=) are used for subsets while the "greater than" symbols (>, >=) are used for supersets.

Less-than and greater-than imply strictness, meaning that the two sets being compared cannot be equal to each other. The equal sign allows for less strict improper subsets and supersets. Sets support both proper (<) and improper (<=) subsets as well as proper (>) and improper (>=) supersets. A set is "less than" another set if and only if the first set is a proper subset of the second set (is a subset but not equal), and a set is "greater than" another set if and only if the first set is a proper superset of the second set (is a superset but not equal).

```
>>> set('shop') < set('cheeseshop')
```

```
True
```

```
>>> set('bookshop') >= set('shop')
```

```
True
```

ii) Set Type Operators (All Set Types)

Union (|)

The union operation is practically equivalent to the OR (or inclusive disjunction) of sets. The union of two sets is another set where each element is a member of at least one of the sets, i.e., a member of one set or the other. The union symbol has a method equivalent, union().

```
>>> s | t
```

```
set(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
```

Intersection (&)

You can think of the intersection operation as the AND (or conjunction) of sets. The intersection of two sets is another set where each element must be a member of at both sets, i.e., a member of one set and the other. The intersection symbol has a method equivalent, intersection() .

```
>>> s & t
set(['h', 's', 'o', 'p'])
```

Difference/Relative Complement (-)

The difference, or relative complement, between two sets is another set where each element is in one set but not the other. The difference symbol has a method equivalent, difference() .

```
>>> s - t
set(['c', 'e'])
```

Symmetric Difference (^)

Similar to the other Boolean set operations, symmetric difference is the XOR (or exclusive disjunction) of sets. The symmetric difference between two sets is another set where each element is a member of one set but not the other. The symmetric difference symbol has a method equivalent, symmetric_difference()

```
() .
>>> s ^ t
set(['k', 'b', 'e', 'c'])
```

Mixed Set Type Operations

In the above examples, s is a set while t is a frozenset. Note that each of the resulting sets from using the set operators above result in sets. However note that the resulting type is different when the operands are reversed:

```
>>> t | s
frozenset(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
>>> t ^ s
frozenset(['c', 'b', 'e', 'k'])
>>> t - s
frozenset(['k', 'b'])
```

iii) Set Type Operators (Mutable Sets Only)

(Union) Update (| =)

The update operation adds (possibly multiple) members from another set to the existing set. The method equivalent is `update()` .

```
>>> s = set('cheeseshop')
>>> u = frozenset(s)
>>> s |= set('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
```

Retention/Intersection Update (& =)

The retention (or intersection update) operation keeps only the existing set members that are also elements of the other set. The method equivalent is `intersection_update()` .

```
>>> s = set(u)
>>> s &= set('shop')
>>> s
set(['h', 's', 'o', 'p'])
```

Difference Update (- =)

The difference update operation returns a set whose elements are members of the original set after removing elements that are (also) members of the other set. The method equivalent is `difference_update()` .

```
>>> s = set(u)
>>> s -= set('shop')
>>> s
set(['c', 'e'])
```

Symmetric Difference Update (^ =)

The symmetric difference update operation returns a set whose members are either elements of the original or other set but not both. The method equivalent is `symmetric_difference_update()` .

```
>>> s = set(u)
>>> t = frozenset('bookshop')
>>> s ^= t
```

```
>>> s
```

```
set(['c', 'b', 'e', 'k'])
```

B) Built-in Functions

i) Standard Type Functions

len()

The len() BIF for sets returns cardinality (or the number of elements) of the set passed in as the argument.

```
>>> s = set(u)
```

```
>>> s
```

```
set(['p', 'c', 'e', 'h', 's', 'o'])
```

```
>>> len(s)
```

```
6
```

ii) Set Type Factory Functions

set() and frozenset()

The set() and frozenset() factory functions generate mutable and immutable sets, respectively. If no argument is provided, then an empty set is created. If one is provided, it must be an iterable, i.e., a sequence, an iterator, or an object that supports iteration such as a file or a dictionary.

```
>>> set()
```

```
set([])
```

```
>>> set([])
```

```
set([])
```

```
>>> set(())
```

```
set([])
```

```
>>> set('shop')
```

```
set(['h', 's', 'o', 'p'])
```

```
>>>
```

```
>>> frozenset(['foo', 'bar'])
```

```
frozenset(['foo', 'bar'])
```

```
>>>
```

```

>>> f = open('numbers', 'w')
>>> for i in range(5):
...
f.write('%d\n' % i)
...
>>> f.close()
>>> f = open('numbers', 'r')
>>> set(f)
set(['0\n', '3\n', '1\n', '4\n', '2\n'])
>>> f.close()

```

C) Set Type Built-in Methods

i) Methods (All Set Types)

We have seen the operator equivalents to most of the built-in methods

Set Type Methods

Method Name Operation

s.issubset(t)	Returns True if every member of s is in t , False otherwise
s.issuperset(t)	Returns true if every member of s is in t , False otherwise
s.union(t)	Returns a new set with the members of s or t
s.intersection(t)	Returns a new set with members of s and t
s.difference(t)	Returns a new set with members of s but not t
s.symmetric_difference(t)	Returns a new set with members of s or t but not both
s.copy()	Returns a new set that is a (shallow) copy of s

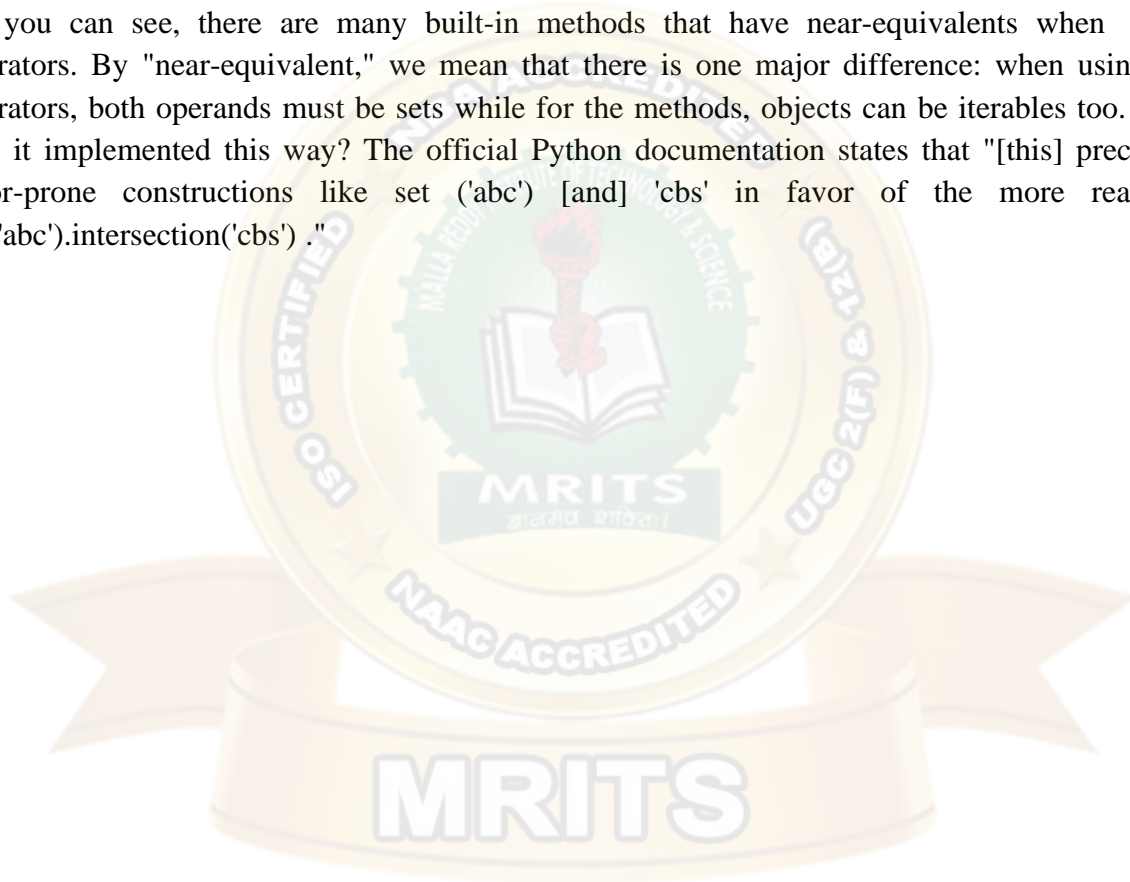
The one method without an operator equivalent is copy() . Like the dictionary method of the same name, it is faster to create a copy of the object using copy() than it is using a factory function like set() ,frozenset() , or dict() .

ii)Methods (Mutable Sets Only)Mutable Set Type Methods

Method Name	Operation
s.update(t)	Updates s with elements added from t ; in other words, s now has members of either s or t
s.intersection_update(t)	Updates s with members of both s and t
s.difference_update(t)	Updates s with members of s without elements of t

iii)Using Operators versus Built-in Methods

As you can see, there are many built-in methods that have near-equivalents when using operators. By "near-equivalent," we mean that there is one major difference: when using the operators, both operands must be sets while for the methods, objects can be iterables too. Why was it implemented this way? The official Python documentation states that "[this] precludes error-prone constructions like `set('abc') [and] 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`."



UNIT – II

FILES: File Objects, File Built-in Function [open()], File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments, File System, File Execution, Persistent Storage Modules, Related Modules

Exceptions: Exceptions in Python, Detecting and Handling Exceptions, Context Management, *Exceptions as Strings, Raising Exceptions, Assertions, Standard Exceptions, *Creating Exceptions, Why Exceptions (Now)?, Why Exceptions at All?, Exceptions and the sys Module, Related Modules

Modules: Modules and Files, Namespaces, Importing Modules, Importing Module Attributes, Module Built-in Functions, Packages, Other Features of Modules

FILES

1. What is a File? Write a short note on File Objects.

File is a named location on disk to store related information, as the file is having some name and location, it is stored in hard disk.

In python, file operation is performed in following order:

- Opening a file.
- Read or Write operation.
- Closing a file.

In python, whenever we try read or write files we don't need to import any library as it's handled natively. The very first thing we'll do is to use the built-in open function to get a file object.

The open function opens a file and returns a file object.

File Objects

A file object allows us to use, access and manipulate all the user accessible files. The file object contains methods and attributes which latter can be used to retrieve information or manipulate the file you opened. File objects can be used not only to access normal disk files, but also any other type of "file" that uses that abstraction. Once the proper "hooks" are installed, you can access other objects with file-like interfaces in the same manner you would access normal files.

The open() built-in function (see below) returns a file object which is then used for all succeeding operations on the file in question. There are a large number of other functions which return a file or file-like object. One primary reason for this abstraction is that many input/output data structures prefer to adhere to a common interface. It provides consistency in behavior as well as implementation. Operating systems like Unix even feature files as an underlying and architectural interface for communication. Remember, files are simply a contiguous sequence of bytes. Anywhere data needs to be sent usually involves a byte stream of some sort, whether the stream occurs as individual bytes or blocks of data.

2. Explain about File Built-in Function.

File Built-in Function [open()]

As the key to opening file doors, the open() built-in function provides a general interface to initiate the file input/output (I/O) process. open() returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or *raises* an IOError exception.

The basic syntax of the `open()` built-in function is:

```
file_object = open(file_name, access_mode='r', buffering=-1)
```

The *file_name* is a string containing the name of the file to open. It can be a relative or absolute/full pathname. The *access_mode* optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes "r," "w," or "a," representing read, write, and append, respectively.

Any file opened with mode "r" must exist. It opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. Any file opened with "w" is for writing only. It overwrites the file if the file exists. If the file does not exist, creates a new file for writing. Any file opened with "a" will be opened for appending. If the file exists, the initial position for file (write) access is set to the end-of-file. If the file does not exist, it will be created, making it the same as if you opened the file in "w" mode.

There are other modes supported by `fopen()` that will work with Python's `open()`. These include the "+" for read-write access and "b" for binary access. One note regarding the binary flag: "b" is antiquated on all Unix systems which are POSIX-compliant (including Linux) because they treat all files as "binary" files, including text files. Here is an entry from the Linux manual page for `fopen()`, which is from where the Python `open()` function is derived:

The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI C3.159-1989 ("ANSI C") and has no effect; the "b" is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

You will find a complete list of file access modes, including the use of "b" if you choose to use it, in [Table9.1](#). If *access_mode* is not given, it defaults automatically to "r."

The other optional argument, *buffering*, is used to indicate the type of buffering that should be performed when accessing the file. A value of 0 means no buffering should occur, a value of 1 signals line buffering, and any value greater than 1 indicates buffered I/O with the given value as the buffer size. The lack of or a negative value indicates that the system default buffering scheme should be used, which is line buffering for any teletype or tty-like device and normal buffering for everything else. Under normal circumstances, a *buffering* value is not given, thus using the system default.

<i>File Mode</i>	<i>Operation</i>
r	open for read
w	open for write (truncate if necessary)
a	open for write (start at EOF, create if necessary)
r+	open for read and write
w+	open for read and write (see "w" above)
a+	open for read and write (see "a" above)
rb	open for binary read
wb	open for binary write (see "w" above)
ab	open for binary append (see "a" above)
rb+	open for binary read and write (see "r+" above)
wb+	open for binary read and write (see "w+" above)
ab+	open for binary read and write (see "a+" above)

Table 9.1. Access Modes for File Objects

Here are some examples for opening files:

```
fp = open('/etc/motd')           #open file for read
fp = open('test','w')          #open file for write
fp = open('data','r+')         #open file for read/write
fp = open('c:\io.sys', 'rb')   #open binary file for read
```

3. Write about File Built-in Methods.

File Built-in Methods

Once open() has completed successfully and returned a file object, all subsequent access to the file transpires with that "handle." File methods come in four different categories: input, output, movement within a file, which we will call "intra-file motion," and miscellaneous.

Input

The read() method is used to read bytes directly into a string, reading at most the number of bytes indicated. If no size is given, the default value is set to -1, meaning that the file is read to the end.

Syntax: `fileObject.read(size);`

size – This is the number of bytes to be read from the file.

The readline() method reads one line of the open file (reads all bytes until a NEWLINE character is encountered). The NEWLINE character is retained in the returned string. The readlines() method is similar, but reads all remaining lines as strings and returns a list containing the read set of lines. The readinto() method reads the given number of bytes into a writable buffer object, the same type of object returned by the unsupported buffer() built-in function. (Since buffer() is not supported, neither is readinto()).

Example:

```
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]
file1.write("Hello \n")
file1.writelines(L)
file1.close()
```



```
file1 = open("myfile.txt","r+")
print "Output of Read function is "
print file1.read()
print
file1.seek(0)
print "Output of Readline function is "
print file1.readline()
print
file1.seek(0)
print "Output of Read(9) function is "
print file1.read(9)
print
file1.seek(0)
print "Output of Readline(9) function is "
print file1.readline(9)
file1.seek(0)
print "Output of Readlines function is "
print file1.readlines()
print
file1.close()
```

Result:

```
Output of Read function is
Hello
This is Delhi
This is Paris
This is London
Output of Readline function is
Hello
Output of Read(9) function is
Hello
Th
Output of Readline(9) function is
Hello
Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

Output

The write() built-in method has the opposite functionality as read() and readline(). It inserts the string in a single line in the text file.

File_object.write(str1)

The writelines () operates on a list just like readlines(), but takes a list of strings and writes them out to a file. NEWLINE characters are not inserted between each line; so if desired, they must be added to the end of each line before writelines()is called.

File_object.writelines(L) for L = [str1, str2, str3]

Note that there is no "writeline()" method since it would be equivalent to calling write() with a single line string terminated with a NEWLINE character.

Intra-file Motion

The seek() method (analogous to the fseek() function in C) moves the file pointer to different positions within the file. The offset in bytes is given along with a *relative offset* location called *whence*. A value of 0 indicates distance from the beginning of a file (note that a position measured from the beginning of a file is also known as the *absolute offset*), a value of 1 indicates movement from the current location in the file, and a value of 2 indicates that the offset is from the end of the file. If you have used fseek() as a C programmer, the values 0, 1, and 2 correspond directly to the constants SEEK_SET, SEEK_CUR, and SEEK_END, respectively. Use of the seek() method comes to play when opening a file for read and write access.

tell() is a complementary method to seek(); it tells you the current location of the file in bytes from the beginning of the file.

Others

The close() method completes access to a file by closing it. The Python garbage collection routine will also close a file when the file object reference has decreased to zero. One way this can happen is when only one reference exists to a file, say, fp = open(), and fp is reassigned to another file object before the original file is explicitly closed. Good programming style suggests closing the file before reassignment to another file object.

The fileno() method passes back the file descriptor to the open file. This is an integer argument that can be used to request I/O operations from the operating system. The flush() method in Python file handling clears the internal buffer of the file. isatty() is a Boolean built-in method that returns true if the file is connected to a tty(-like) device, else False. "tty" originally meant "teletype" and tty(-like) device is any device that acts like a teletype, i.e a terminal. The truncate() method truncates the file's size to 0 or the given size bytes, if the optional size argument is present. To truncate the file, you can open the file in append mode or write mode.

File Method Miscellany

We will now reprise our first file example

```
filename = raw_input('Enter file name: ')
file = open(filename,'r')
allLines = file.readlines()
file.close()
for eachLine in allLines:
    print eachline,
```

We originally described how this program differs from most standard file access in that all the lines are read ahead of time before any display to the screen occurs. Obviously, this is not advantageous if the file is large. In those cases, it may be a good idea to go back to the tried-and-true way of reading and displaying one line at a time:

```
filename = raw_input('Enter file name: ')

```



```

file = open(filename,'r')
done = 0
while not done:
aLine = file.readline()
if aLine != " ":
print aLine,
else:
done = 1
file.close()

```

In this example, we do not know when we will reach the end of the file, so we create a Boolean flag `done`, which is initially set for false. When we reach the end of the file, we will reset this value to true so that the while loop will exit. We change from using `readlines()` to read all lines to `readline()`, which reads only a single line. `readline()` will return a blank line if the end of the file has been reached. Otherwise, the line is displayed to the screen.

We anticipate a burning question you may have... "Wait a minute! What if I have a blank line in my file? Will Python stop and think it has reached the end of my file?" The answer is, of course, no. A blank line in your file will not come back as a blank line. Recall that every line has one or more line separator characters at the end of the line, so a "blank line" would consist of a NEWLINE character or whatever your system uses. So even if the line in your text file is "blank," the line which is read is *not* blank, meaning your application would not terminate until it reaches the end-of-file.

NOTE

One of the inconsistencies of operating systems is the line separator character which their file systems support. On Unix, the line separator is the NEWLINE (\n) character. For the Macintosh, it is the RETURN (\r), and DOS and Windows uses both (\r\n). Check your operating system to determine what your line separator(s) are.

Other differences include the file pathname separator (Unix uses '/', DOS and Windows use '\', and the Macintosh uses ':'), the separator used to delimit a set of file pathnames, and the denotations for the current and parent directories.

These inconsistencies generally add an irritating level of annoyance when creating applications that run on all three platforms (and more if more architectures and operating systems are supported).

Fortunately, the designers of the `os` module in Python have thought of this for us. The `os` module has five attributes which you may find useful. They are listed below in [Table 9.2](#).

os Module Attribute	Description
<code>linesep</code>	string used to separate lines in a file
<code>sep</code>	string used to separate file pathname components
<code>pathsep</code>	string used to delimit a set of file pathnames
<code>curdir</code>	string name for current working directory
<code>pardir</code>	string name for parent (of current working directory)

Table 9.2. osModule Attributes to Aid in Multi-platform Development

We would also like to remind you that the comma placed at the end of the **print** statement is to suppress the NEWLINE character that **print** normally adds at the end of output. The reason for this is

because every line from the text file already contains a NEWLINE. `readline()` and `readlines()` do not strip off any whitespace characters in your line. If we omitted the comma, then your text file display would be doublespaced one NEWLINE which is part of the input and another added by the **print** statement.

Before moving on to the next section, we will show two more examples, the first highlighting output to files (rather than input), and the second performing both file input and output as well as using the `seek()` and `tell()` methods for file positioning.

```
filename = raw_input('Enter file name: ')
file = open(filename,'w') done = 0
while not done:
    aLine = raw_input("Enter a line ( '.' to quit): ")
    if aLine != ".": file.write(aLine + '\n')
    else:
        done = 1 file.close()
```

This piece of code is practically the opposite of the previous. Rather than reading one line at a time and displaying it, we ask the user for one line at a time, and send them out to the file. Our call to the `write()` method must contain a NEWLINE because `raw_input()` does not preserve it from the user input. Because it may not be easy to generate an end-of-file character from the keyboard, the program uses the period (.) as its end-of-file character, which, when entered by the user, will terminate input and close the file.

Our final example opens a file for read and write, creating the file `scratch` (after perhaps truncating an already-existing file). After writing data to the file, we move around within the file using `seek()`. We also use the `tell()` method to show our movement.

```
>>> f = open('/tmp/x', 'w+')
>>> f.tell() 0
>>> f.write('test line 1\n')      # add 12-char string [0-11]
>>> f.tell() 12
>>> f.write('test line 2\n')     # add 12-char string [12-23]
>>> f.tell()                    # tell us current file location (end) 24
>>> f.seek(-12, 1)              # move back 12 bytes
>>> f.tell()                    # to beginning of line 2 12
>>> f.readline() 'test line 2\n'
>>> f.seek(0, 0)                # move back to beginning
>>> f.readline() 'test line 1\n'
>>> f.tell()                    # back to line 2 again 12
>>> f.readline() 'test line 2\n'
>>> f.tell()                    # at the end again 24
>>> f.close()                   # close file
```

[Table 9.3](#) lists all the built-in methods for file objects:

Table 9.3. Methods for File Objects

<i>File Object Method</i>	<i>Operation</i>
<code>file.close()</code>	close <i>file</i>
<code>file.fileno()</code>	return integer file descriptor (FD) for <i>file</i>
<code>file.flush()</code>	flush internal buffer for <i>file</i>
<code>file.isatty()</code>	return 1 if <i>file</i> is a tty-like device, 0 otherwise
<code>file.read(size=-1)</code>	read all or <i>size</i> bytes of file as a string and return it
<code>file.readinto(buf, size)</code>	read <i>size</i> bytes from <i>file</i> into buffer <i>buf</i>
<code>file.readline()</code>	read and return one line from <i>file</i> (includes trailing "\n")
<code>file.readlines()</code>	read and returns all lines from <i>file</i> as a list (includes all trailing "\n" characters)
<code>file.seek(off, whence)</code>	move to a location within <i>file</i> , <i>off</i> bytes offset from <i>whence</i> (0 == beginning of file, 1 == current location, or 2 == end of file)
<code>file.tell()</code>	return current location within <i>file</i>
<code>file.truncate(size=0)</code>	truncate <i>file</i> to 0 or <i>size</i> bytes
<code>file.write(str)</code>	write string <i>str</i> to <i>file</i>
<code>file.writelines(list)</code>	write <i>list</i> of strings to <i>file</i>

4. What are the File Built-in Attributes?

File Built-in Attributes

File objects also have data attributes in addition to its methods. These attributes hold auxiliary data related to the file object they belong to, such as the file name (*file.name*), the mode with which the file was opened (*file.mode*), whether the file is closed (*file.closed*), and a flag indicating whether an additional space character needs to be displayed before successive data items when using the **print** statement (*file.softspace*). [Table 9.4](#) lists these attributes along with a brief description of each.

<i>File Object Attribute</i>	<i>Description</i>
<code>file.closed</code>	1 if <i>file</i> is closed, 0 otherwise
<code>file.mode</code>	access mode with which <i>file</i> was opened
<code>file.name</code>	name of <i>file</i>
<code>file.softspace</code>	0 if space explicitly required with print , 1 otherwise; rarely used by the programmer—generally for internal use only

Table 9.4. Attributes for File Objects

5. Discuss about Standard Files in Python.

There are generally three standard files which are made available to you when your program starts. These are standard input (usually the keyboard), standard output (buffered output to the monitor or display), and standard error (unbuffered output to the screen). (The "buffered" or "unbuffered" output refers to that third argument to `open()`). These files are named `stdin`, `stdout`, and `stderr` and take after their names from the C language. When we say these files are "available to you when your program starts," that means that these files are pre-opened for you, and access to these files may commence once you have their file handles.

Python makes these file handles available to you from the `sys` module. Once you import `sys`, you have access to these files as `sys.stdin`, `sys.stdout`, and `sys.stderr`. The **print** statement normally outputs to

sys.stdout while the raw_input() built-in function receives its input from sys.stdin.

Python raw_input function is used to get the values from the user. We call this function to tell the program to stop and wait for the user to input the values. It is a built-in function.

We will now take yet another look at the "Hello World!" program so that you can compare the similarities and differences between using **print**/raw_input() and directly with the file names:

print

```
print 'Hello World!'
```

sys.stdout.write()

```
import sys
```

```
sys.stdout.write('Hello World!' + '\n')
```

Notice that we have to explicitly provide the NEWLINE character to sys.stdout's write() method. In the input examples below, we do not because readline() executed on sys.stdin preserves the newline. raw_input() does not, hence we will allow print to add its NEWLINE.

raw_input()

```
aString = raw_input('Enter a string: ')
```

```
print aString
```

sys.stdin.readline()

```
import sys
```

```
sys.stdout.write('Enter a string: ')
```

```
aString = sys.stdin.readline()
```

```
sys.stdout.write(aString)
```

6. Explain about the Command-line Arguments.

Command-line arguments allow a programmer or administrator to start a program perhaps with different behavioral characteristics. Much of the time, this execution takes place in the middle of the night and run as a batch job without human interaction. Command-line arguments and program options enable this type of functionality. As long as there are computers sitting idle at night and plenty of work to be done, there will always be a need to run programs in the background on our very expensive "calculators."

The sys module also provides access to any *command-line arguments* via the sys.argv. Command-line arguments are those arguments given after the name of the program in the command line shell of the operating system. Historically, of course, these arguments are so named because they are given on the command-line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case.

Most IDEs provide a separate window with which to enter your "command-line arguments." These, in turn, will be passed into the program as if you started your application from the command-line.

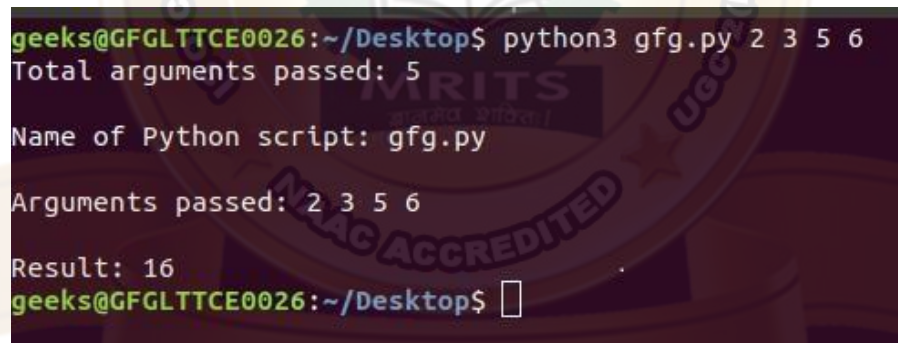
The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. One such variable is sys.argv which is a simple list structure. Its main purpose is:

- It is a list of command line arguments.
- `len(sys.argv)` provides the number of command line arguments.
- `sys.argv[0]` is the name of the current Python script.

Let us create a small test program for adding two numbers and the numbers are passed as command-line arguments.

```
import sys
n = len(sys.argv)
print("Total arguments passed:", n)
print("\nName of Python script:", sys.argv[0])
print("\nArguments passed:", end = " ")
for i in range(1, n):
    print(sys.argv[i], end = " ")
Sum = 0
for i in range(1, n):
    Sum += int(sys.argv[i])
print("\n\nResult:", Sum)
```

Output:



```
geeks@GFGLTCE0026:~/Desktop$ python3 gfg.py 2 3 5 6
Total arguments passed: 5
Name of Python script: gfg.py
Arguments passed: 2 3 5 6
Result: 16
geeks@GFGLTCE0026:~/Desktop$
```

Are command-line arguments useful? Unix commands are typically programs which take input, perform some function, and send output as a stream of data. This data is usually sent as input directly to the next program, which some other types of function or calculation and sends the new output to another program, and so on. Rather than saving the output of each program and potentially taking up a good amount of disk space, the output is usually "piped" in to the next program as *its* input. This is accomplished by providing data on the command-line or through standard input. When a program displays or sends output to the standard output file, the result would be displayed on the screen—unless that program is also "piped" to another program, in which case that standard output file is really the standard input file of the next program. Python features a `getopt` module that helps you parse command-line options and arguments.

7. What is a File System in Python? Write a short note on it.

Access to your file system occurs mostly through the Python `os` module. This module serves as the primary interface to your operating system facilities and services from Python. The `os` module is actually a front-end to the real module that is loaded, a module that is clearly operating system-

dependent. This "real" module may be one of the following: posix (Unix), nt (Windows), mac (Macintosh), dos (DOS), os2 (OS/2), etc. You should never import those modules directly. Just import os and the appropriate module will be loaded, keeping all the underlying work hidden from sight. Depending on what your system supports, you may not have access to some of the attributes which may be available in other operating system modules.








In addition to managing processes and the process execution environment, the os module performs most of the major file system operations that the application developer may wish to take advantage of. These features include removing and renaming files, traversing the directory tree, and managing file accessibility. [Table 9.5](#) lists some of the more common file or directory operations available to you from the os module.

A second module that performs specific pathname operations is also available. The os.path module is accessible through the os module. Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries. [Table 9.6](#) outlines some of the more common functions in os.path.

These two modules allow for consistent access to the file system regardless of platform or operating system. The program in [Example 9.1](#) (ospathex.py) test drives some of these functions from the os and os.path modules.

<i>os Module File/Directory Function</i>	<i>Operation</i>
File Processing	
remove()/unlink()	delete file
rename()	rename file
*stat() ^[a]	return file statistics
symlink()	create symbolic link
utime()	update timestamp
Directories/Folders	
chdir()	change working directory
listdir()	list files in directory
getcwd()	return current working directory
mkdir()/makedirs()	create directory(ies)
rmdir()/removedirs()	remove directory(ies)

Table 9.5. osModule File/Directory Access Functions

<i>Access/Permissions (available only on Unix  or Windows )</i>	
access()	verify permission modes 
chmod()	change permission modes  
umask()	set default permission modes  

[a] includes stat(), lstat(), xstat()

Table 9.6. os.path Module Pathname Access Functions

<code>os.path</code> Pathname Function	Operation
Separation	
<code>basename()</code>	remove directory path and return leaf name
<code>dirname()</code>	remove leaf name and return directory path
<code>join()</code>	join separate components into single pathname
<code>split()</code>	return (<code>dirname()</code> , <code>basename()</code>) tuple
<code>splitdrive()</code>	return (<code>drivename</code> , <code>pathname</code>) tuple
<code>splittext()</code>	return (<code>filename</code> , <code>extension</code>) tuple
Information	
<code>getatime()</code>	return last file access time
<code>getmtime()</code>	return last file modification time
<code>getsize()</code>	return file size (in bytes)
Inquiry	
<code>exists()</code>	does pathname (file or directory) exist?
<code>isdir()</code>	does pathname exist and is a directory?
<code>isfile()</code>	does pathname exist and is a file?
<code>islink()</code>	does pathname exist and is a symbolic link?
<code>samefile()</code>	do both pathnames point to the same file?

Example 9.1. `os` & `os.path` Modules Example (`ospathex.py`)

This code exercises some of the functionality found in the `os` and `os.path` modules. It creates a test file, populates a small amount of data in it, renames the file, and dumps its contents. Other auxiliary file operations are performed as well, mostly pertaining to directory tree traversal and file pathname manipulation.

```

<$npage>
001 1  #!/usr/bin/env python
002 2
003 3  import os
004 4  for tmpdir in ('/tmp', 'c:/windows/temp'):
005 5  if os.path.isdir(tmpdir):
006 6  break
007 7  else: <$npage>
008 8  print 'no temp directory available'
009 9  tmpdir = "
010 10
011 11 if tmpdir:
012 12     os.chdir(tmpdir)
013 13     cwd = os.getcwd()
014 14     print '*** current temporary directory'
015 15     print cwd
016 16
017 17     print '*** creating example directory...'
018 18     os.mkdir('example')
019 19     os.chdir('example')
020 20     cwd = os.getcwd()
021 21     print '*** new working directory:'
022 22     print cwd

```

```
023 23     print '*** original directory listing:'
024 24     print os.listdir(cwd)
025 25
026 26     print '*** creating test file...'
027 27     file = open('test', 'w')
28 28     file.write('foo\n')
29 29     file.write('bar\n')
30 30     file.close()
31 31     print '*** updated directory listing:'
032 32     print os.listdir(cwd)
033 33
034 34     print "*** renaming 'test' to 'filetest.txt'"
035 35     os.rename('test', 'filetest.txt')
036 36     print '*** updated directory listing:'
037 37     print os.listdir(cwd)
038 38
039 39     path = os.path.join(cwd, os.listdir (cwd)[0])
040 40     print     '*** full file pathname'
041 41     print     path
042 42     print     '*** (pathname, basename) =='
043 43     print     os.path.split(path)
044 44     print     '*** (filename, extension) =='
045 45     print     os.path.splitext(os.path.basename
                                (path))
046 46
047 47     print     '*** displaying file contents:'
048 48     file = open(path)
049 49     allLines = file.readlines()
050 50     file.close()
051 51     for eachLine in allLines:
052 52     print eachLine,
053 53
054 54     print '*** deleting test file'
055 55     os.remove(path)
056 56     print '*** updated directory listing:'
057 57     print os.listdir(cwd)
058 58     os.chdir(os.pardir)
059 59     print '*** deleting test directory'
060 60     os.rmdir('example')
061 61     print '*** DONE'
062     <$nopage>
```

Running this program on a Unix platform, we get the following output:

```
% ospathex.py
*** current temporary directory
/tmp
*** creating example directory...
*** new working directory:
/tmp/example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing: ['filetest.txt']
*** full file pathname:
/tmp/example/filetest.txt
*** (pathname, basename) == ('/tmp/example', 'filetest.txt')
*** (filename, extension) == ('filetest', '.txt')
*** displaying file contents:
foo bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```

Running this example from a DOS window results in very similar execution:

```
C:\>python ospathex.py
*** current temporary directory c:\windows\temp
*** creating example directory...
*** new working directory: c:\windows\temp\example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing: ['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing: ['filetest.txt']
*** full file pathname: c:\windows\temp\example\filetest.txt
*** (pathname, basename) == ('c:\\windows\\temp\\example', 'filetest.txt')
*** (filename, extension) == ('filetest', '.txt')
*** displaying file contents:
foo bar
*** deleting test file
*** updated directory listing:
[]
```

*** deleting test directory

*** DONE

Rather than providing a line-by-line explanation here, we will leave it to the reader as an exercise. However, we will walk through a similar interactive example (including errors) to give you a feel for what it is like to execute this script one step at a time. We will break into the code every now and then to describe the code we just encountered.

```
>>> import os
>>> os.path.isdir('/tmp') 1
>>> os.chdir('/tmp')
>>> cwd = os.getcwd()
>>> cwd '/tmp'
```

This first block of code consists of importing the os module (which also grabs the os.path module). We verify that '/tmp' is a valid directory and change to that temporary directory to do our work. When we arrive, we call the getcwd() method to tell us where we are.

```
>>> os.mkdir('example')
>>> cwd = os.getcwd()
>>> cwd '/tmp/example'
>>>
>>> os.listdir()
Traceback (innermost last): File "<stdin>", line 1, in ?
TypeError: function requires at least one argument
>>>
>>> os.listdir(cwd) []
```

Next, we create a subdirectory in our temporary directory, after which we will use the listdir() method to confirm that the directory is indeed empty (since we just created it). The problem with our first call to listdir() was that we forgot to give the name of the directory we want to list. That problem is quickly remedied on the next line of input.

```
>>> file = open('test', 'w')
>>> file.write('foo\n')
>>> file.write('bar\n')
>>> file.close()
>>> os.listdir(cwd) ['test']
```

We then create a test file with two lines and verify that the file has been created by listing the directory again afterwards.

```
>>> os.rename('test', 'filetest.txt')
>>> os.listdir(cwd) ['filetest.txt']
>>>
>>> path = os.path.join(cwd, os.listdir(cwd)[0])
>>> path '/tmp/example/filetest.txt'
>>>
>>> os.path.isfile(path) 1
>>> os.path.isdir(path) 0
>>>
```



```
>>> os.path.split(path) ('/tmp/example', 'filetest.txt')
>>>
>>> os.path.splitext(os.path.basename(path)) ('filetest', '.ext')
```

This section is no doubt an exercise of os.path functionality, testing join(), isfile(), isdir() which we have seen earlier, split(), basename(), and splitext(). We also call the rename() function from os.

```
>>> file = open(path)
>>> file.readlines()
>>> file.close()
>>>
>>> for eachLine in allLines:
...     print eachLine,
...     foo bar
```

This next piece of code should be familiar to the reader by now, since this is the third time around. We open the test file, read in all the lines, close the file, and display each line, one at a time.

```
>>> os.remove(path)
>>> os.listdir(cwd) []
>>> os.chdir(os.pardir)
>>> os.rmdir('example')
```

This last segment involves the deletion of the test file and test directory concluding execution. The call to chdir() moves us back up to the main temporary directory where we can remove the test directory (os.pardir contains the parent directory string ".." for Unix and Windows; the Macintosh uses "::"). It is not advisable to remove the directory that you are in.

File Execution

Whether we want to simply run an operating system command, invoke a binary executable, or another type of script (perhaps a shell script, Perl, or Tcl/Tk), this involves executing another file somewhere else on the system. Even running other Python code may call for starting up another Python interpreter, although that may not always be the case.

8. Depict the various ways to archive python objects using Persistent Storage Modules.

In many of the exercises in this text, user input is required for those applications. After many iterations, it may be somewhat frustrating being required to enter the same data repeatedly. The same may occur if you are entering a significant amount of data for use in the future.

This is where it becomes useful to have persistent storage, or a way to archive your data so that you may access it at a later time instead of having to re-enter all of that information.

When simple disk files are no longer acceptable and full relational database management systems (RDBMSs) are overkill, simple persistent storage fills the gap. The majority of the persistent storage modules deals with storing strings of data, but there are ways to archive Python objects as well.

Pickle and marshal Modules

Python provides a variety of modules which implement minimal persistent storage. One set of modules (marshal and pickle) allows for pickling of Python objects. Pickling is the process whereby

objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms. Pickling is also known as flattening, serializing, or marshalling.

Another set of modules (dbhash/bsddb, dbm, gdbm, dumbdbm) and their "manager" (anydbm) can provide persistent storage of Python strings only. The last module (shelve) can do both.

As we mentioned before, both marshal and pickle can flatten Python objects. These modules do not provide "persistent storage" per se, since they do not provide a namespace for the objects, nor can they provide concurrent write access to persistent objects. What they can do, however, is to pickle Python objects to allow them to be stored or transmitted.

Storage, of course, is sequential in nature (you store or transmit objects one after another). The difference between marshal and pickle is that marshal can handle only simple Python objects (numbers, sequences, mapping, and code) while pickle can transform recursive objects, objects that are multi-referenced from different places, and user-defined classes and instances. The pickle module is also available in a turbo version called cPickle, which implements all functionality in C.

DBM-style Modules

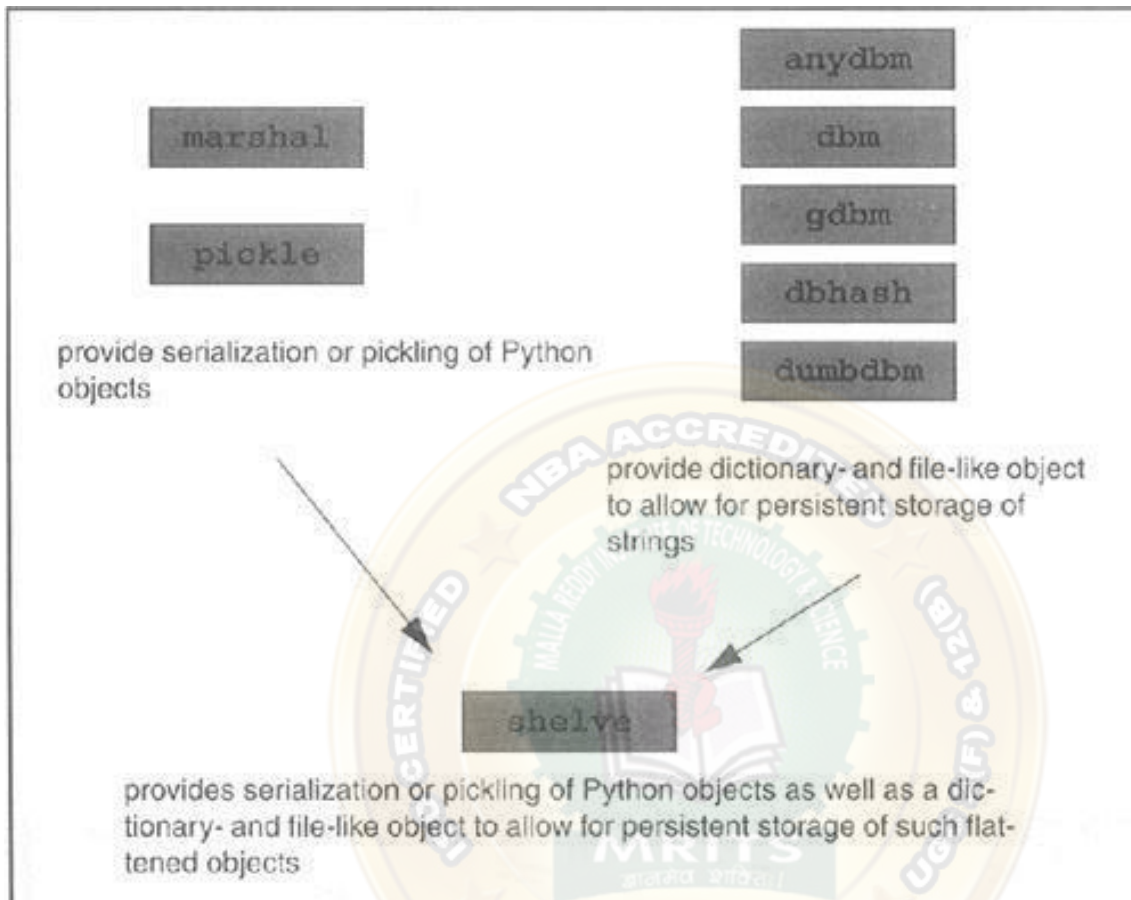
The *db* series of modules writes data in the traditional DBM format. There are a large number of different implementations: dbhash/bsddb, dbm, gdbm, and dumbdbm. We highly recommend the use of the anydbm module, which detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal. The dumbdbm module is the most limited one, and is the default used if none of the other packages are available. These modules do provide a namespace for your objects, using objects which behave similar to a combination of a dictionary object and a file object.

The one limitation of these systems is that they can store only strings. In other words, they do not serialize Python objects.

Shelve Module

Finally, we have a somewhat more complete solution, the shelve module. The shelve module uses the anydbm module to find a suitable DBM module, then uses cPickle to perform the pickling process. The shelve module permits concurrent read access to the database file, but not shared read/write access. This is about as close to persistent storage as you will find in the Python standard library. There may other external extension modules which implement "true" persistent storage. The diagram in [Figure 9-1](#) shows the relationship between the pickling modules and the persistent storage modules, and how the shelve object appears to be the best of both worlds.

Figure 9.1. Python Modules for Serialization and Persistency



Related Modules

There are plenty of other modules related to files and input/output, all of which work on most of the major platforms. [Table 9.7](#) lists some of the file-related modules.

<i>Module(s)</i>	<i>Contents</i>
fileinput	iterates over lines of multiple input text files
getopt	provides command-line argument parsing/manipulation
glob/fnmatch	provides Unix-style wildcard character matching
gzip/zlib/zipfile	allows file access to include automatic de/compression
shutil	offers high-level file access functionality
c/StringIO	implements file-like interface on top of string objects
tempfile	generates temporary file names or files

Table 9.7. Related File Modules

The fileinput module iterates over a set of input files and reads their contents one line at a time, allowing you to iterate over each line, much like the way the Perl (< >) operator works without any provided arguments. File names that are not explicitly given will be assumed to be provided from the command-line.

The glob and fnmatch modules allow for file name pattern-matching in the good old fashioned Unix shell-style, for example, using the asterisk (*) wildcard character for all string matches and the (?) for matching single characters.

The gzip and zlib modules provide direct file access to the zlib compression library. The gzip module, written on top of the zlib module, allows for standard file access, but provides for automatic gzip-compatible compression and decompression. Note that if you are compiling your Python interpreter, you have to enable the zlib module to be built (by editing the Modules/Setup file). It is not turned on by default. The new zipfile module, also requiring the zlib module, allows the programmer to create, modify, and read ziparchive files.

The shutil module furnishes high-level file access, performing such functions as copying files, copying file permissions, and recursive directory tree copying, to name a few.

The tempfile module can be used to generate temporary file names and files.

In our earlier chapter on strings, we described the StringIO module (and its C-compiled companion cStringIO), and how it overlays a file interface on top of string objects. This interface includes all of the standard methods available to regular file objects.

The modules we mentioned in the Persistent Storage section above ([Section 9.9](#)) include examples of a hybrid file- and dictionary-like object.

Some other Python modules which generate file-like objects include network and file socket objects (socket module), the popen*() file objects that connect your application to other running processes (os and popen2 modules), the fdopen() file object used in low-level file access (os module), and opening a network connection to an Internet web server via its Uniform Resource Locator (URL) address (urllib module). Please be aware that not all standard file methods may be implemented for these objects. Likewise, they may provide functionality in addition to what is available for regular files.

Errors and Exceptions

9. What are Exceptions in python? Explain each with suitable example.

Errors

Before we get into details about what exceptions are, let us review what errors are. In the context of software, errors are either syntactical or logical in nature. Syntax errors indicate errors with the construct of the software and cannot be executed by the interpreter or compiler correctly. These errors must be repaired before execution can occur.

Once programs are semantically correct, the only errors which remain are logical. Logical errors can either be caused by lack of or invalid input, or, in other cases, by the logic's not being able to generate, calculate, or otherwise produce the desired results based on the input. These errors are sometimes known as domain and range failures, respectively.

When errors are detected by Python, the interpreter indicates that it has reached a point where continuing to execute in the current flow is no longer possible. This is where exceptions come into the picture.

Exceptions

Exceptions can best be described as action that is taken outside of the normal flow of control because of errors. This action comes in two distinct phases, the first being the error which causes an exception to occur, and the second being the detection (and possible resolution) phase.

The first phase takes place when an *exception condition* (sometimes referred to as *exceptional condition*) occurs. Upon detection of an error and recognition of the exception condition, the interpreter performs an operation called *raising* an exception. Raising is also known as triggering, throwing, or generating, and is the process whereby the interpreter makes it known to the current control flow that something is wrong. Python also supports the ability of the programmer's to raise exceptions. Whether triggered by the Python interpreter or the programmer, exceptions signal that an error has occurred. The current flow of execution is interrupted to process this error and take appropriate action, which happens to be the second phase.

The second phase is where exception handling takes place. Once an exception is raised, a variety of actions can be invoked in response to that exception. These can range anywhere from ignoring the error, logging the error but otherwise taking no action, performing some corrective measures and aborting the program, or alleviating the problem to allow for resumption of execution. Any of these actions represents a *continuation*, or an alternative branch of control. The key is that the programmer can dictate how the program operates when an error occurs.

As you may have already concluded, errors during run-time are primarily caused by external reasons, such as poor input, a failure of some sort, etc. These causes are not under the direct control of the programmer, who can anticipate only a few of the errors and code the most general remedies.

Languages like Python which support the raising and—more importantly—the handling of exceptions empowers the developer by placing them in a more direct line of control when errors occur. The programmer not only has the ability to detect errors, but also to take more concrete and remedial actions when they occur. Due to the ability to manage errors during run-time, application robustness is increased.

Exceptions in Python

Like some of the other languages supporting exception handling, Python is endowed with the concepts of a "try" block and "catching" exceptions and, in addition, provides for more "disciplined" handling of exceptions. By this we mean that you can create different handlers for different exceptions, as opposed to a general "catch-all" code where you may be able to detect the exception which occurred in a post-mortem fashion.

A "traceback" notice appears along with a notice with as much diagnostic information as the interpreter can give you, including the error name, reason, and perhaps even the line number near or exactly where the error occurred. All errors have a similar format, regardless of whether running within the Python interpreter or standard script execution, providing a consistent error interface. All errors, whether they be syntactical or logical, result from behavior incompatible with the Python interpreter and cause exceptions to be raised. Let us take a look at some exceptions now.

NameError:attempt to access an undeclared variable

```
>>> foo
Traceback (innermost last):
File "<interactive input>", line 0, in ?
NameError: foo
```

NameError indicates access to an uninitialized variable. The offending identifier was not found in the Python interpreter's symbol table. We will be discussing *namespaces* in an upcoming chapter, but as an introduction, regard them as "address books" linking names to objects. Any object which is accessible should be listed in a namespace. Accessing a variable entails a search by the interpreter, and if the name requested is not found in any of the namespaces, a NameErrorexception will be generated.

ZeroDivisionError:division by any numeric zero

```
>>> 12.4/0.0
Traceback (innermost last):
File "<interactive input>", line 0, in ?
ZeroDivisionError: float division
```

Our example above used floats, but in general, any numeric division-by-zero will result in a ZeroDivisionErrorexception.

SyntaxError:Python interpreter syntax error

```
>>> for
File "<string>", line 1
for
^
SyntaxError: invalid syntax
```

SyntaxError exceptions are the only ones which do not occur at run-time. They indicate an improperly constructed piece of Python code which cannot execute until corrected. These errors are generated at compile-time, when the interpreter loads and attempts to convert your script to Python bytecode. These may also occur as a result of importing a faulty module.

IndexError:request for an out-of-range index for sequence

```
>>> aList = []
>>> aList[0]
Traceback (innermost last):
File "<stdin>", line 1, in ?
IndexError: list index out of range
```

IndexError is raised when attempting to access an index which is outside the valid range of a sequence.

KeyError:request for a non-existent dictionary key

```
>>> aDict = {'host': 'earth', 'port': 80}
>>> print aDict['server']
Traceback (innermost last):
File "<stdin>", line 1, in ?
KeyError: server
```

Mapping types such as dictionaries depend on keys to access data values. Such values are not retrieved if an incorrect/nonexistent key is requested. In this case, a KeyError is raised to indicate such an incident has occurred.

IOError:input/output error

```
>>> f = open("blah")
Traceback (innermost last):
File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'blah'
```

Attempting to open a non-existent disk file is one example of an operating system input/output (I/O) error. Any type of I/O error raises an IOError exception.

AttributeError:attempt to access an unknown object attribute

```
>>> class myClass:
...     pass
...
>>> myInst = myClass()
>>> myInst.bar = 'spam'
>>> myInst.bar 'spam'
>>> myInst.foo
Traceback (innermost last):
File "<stdin>", line 1, in ?
AttributeError: foo
```

In our example, we stored a value in myInst.bar, the bar attribute of instance myInst. Once an attribute has been defined, we can access it using the familiar dotted-attribute notation, but if it has not, as in our case with the foo (non-)attribute, an AttributeError occurs.

10. How to detect and handle exceptions in python? Explain in detail.

Detecting and Handling Exceptions

Exceptions can be detected by incorporating them as part of a **try** statement. Any code suite of a **try** statement will be monitored for exceptions.

There are two main forms of the **try** statement: **try-except** and **try-finally**. These statements are mutually exclusive, meaning that you pick only one of them. A **try** statement is either accompanied by one or more **except** clauses or exactly one **finally** clause. (There is no such thing as a hybrid "try-except-finally.")

try-except statements allow one to detect and handle exceptions. There is even an optional **else** clause for situations where code needs to run only when no exceptions are detected. Meanwhile, **try-finally** statements allow only for detection and processing of any obligatory clean-up (whether or not exceptions occur), but otherwise has no facility in dealing with exceptions.

try-exceptStatement

The **try-except** statement (and more complicated versions of this statement) allows you to define a section of code to monitor for exceptions and also provides the mechanism to execute handlers for exceptions.

The syntax for the most general **try-except** statement looks like this:

```
try:  
    try_suite      # watch for exceptions here  
except Exception:  
    except_suite # exception-handling code
```

Let us give one example, then explain how things work. We will use our IOError example from above. We can make our code more robust by adding a **try-except** "wrapper" around the code:

```
>>> try:  
...     f = open('blah')  
...     except IOError:  
...         print 'could not open file'  
...  
could not open file
```

As you can see, our code now runs seemingly without errors. In actuality, the same IOError still occurred when we attempted to open the nonexistent file. The difference? We added code to both detect and handle the error. When the IOError exception was raised, all we told the interpreter to do was to output a diagnostic message. The program continues and does not "bomb out" as our earlier example—a minor illustration of the power of exception handling. So what is really happening codewise?

During run-time, the interpreter attempts to execute all the code within the **try** statement. If an exception does not occur when the code block has completed, execution resumes past the **except** statement. When the specified exception named on the **except** statement does occur, control flow immediately continues in the handler (all remaining code in the try clause is skipped). In our example above, we are catching only IOError exceptions. Any other exception will not be caught with the handler we specified. If, for example, you want to catch an OSError, you have to add a handler for that particular exception.

NOTE

The remaining code in the **try** suite from the point of the exception is never reached (hence never executed). Once an exception is raised, the race is on to decide on the continuing flow of control. The remaining code is skipped, and the search for a handler begins. If one is found, the program continues in the handler.

If the search is exhausted without finding an appropriate handler, the exception is then propagated to the caller's level for handling, meaning the stack frame immediately preceding the current one. If there is no handler at the next higher level, the exception is yet again propagated to its caller. If the top level is reached without an appropriate handler, the exception is considered unhandled, and the Python interpreter will display the traceback and exit.

Wrapping a Built-in Function

We will now present an interactive example starting with the bare necessity of detecting an error, then building continuously on what we have to further improve the robustness of our code. The premise is in detecting errors while trying to convert a numeric string to a proper (numeric object) representation of its value.

The `float()` built-in function has a primary purpose of converting any numeric type to a float. In Python 1.5, `float()` was given the added feature of being able to convert a number given in string representation to an actual float value, obsoleting the use of the `atof()` function of the `string` module. Readers with older versions of Python may still use `string.atof()`, replacing `float()`, in the examples we use here.

```
>>> float(12345)
12345.0
>>> float('12345')
12345.0
>>> float('123.45e67')
1.2345e+069
```

Unfortunately, `float()` is not very forgiving when it comes to bad input:

```
>>> float('abcde')
Traceback (innermost last):
File "<stdin>", line 1, in ?
float('abcde')
ValueError: invalid literal for float(): abcde
>>>
>>> float(['this is', 1, 'list'])
Traceback (innermost last):
File "<stdin>", line 1, in ?
float(['this is', 1, 'list'])
TypeError: object can't be converted to float
```

Notice in the errors above that `float()` does not take too kindly to strings which do not represent numbers or non-strings. Specifically, if the correct argument type was given (string type) but that type contained an invalid value, the exception raised would be `ValueError` because it was the value that was improper, not the type. In contrast, a list is a bad argument altogether, not even being of the correct type; hence, `TypeError` was thrown.

Our exercise is to call `float()` "safely," or in a more "safe manner," meaning that we want to ignore

error situations because they do not apply to our task of converting numeric string values to floating point numbers, yet are not severe enough errors that we feel the interpreter should abandon execution. To accomplish this, we will create a "wrapper" function, and, with the help of **try-except**, create the environment that we envisioned. We shall call it `safe_float()`. In our first iteration, we will scan and ignore only `ValueErrors`, because they are the more likely culprit. `TypeError`s rarely happen since somehow a non-string must be given to `float()`.

```
def safe_float(object):
    try:
        return float(object)
    except ValueError:
        pass
```

The first step we take is to just "stop the bleeding." In this case, we make the error go away by just "swallowing it." In other words, the error will be detected, but since we have nothing in the **except** suite (except the **pass** statement, which does nothing but serve as a syntactical placeholder for where code is supposed to go), no handling takes place. We just ignore the error.

One obvious problem with this solution is that we did not explicitly return anything to the function caller in the error situation. Even though `None` is returned (when a function does not return any value explicitly, i.e., completing execution without encountering a **return object** statement), we give little or no hint that anything wrong took place. The very least we should do is to explicitly return `None` so that our function returns a value in both cases and makes our code somewhat easier to understand:

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = None
    return retval
```

Bear in mind that with our change above, nothing about our code changed except that we used one more local variable. In designing a well-written application programmer interface (API), you may have kept the return value more flexible. Perhaps you documented that if a proper argument was passed to `safe_float()`, then indeed, a floating point number would be returned, but in the case of an error, you chose to return a string indicating the problem with the input value. We modify our code one more time to reflect this change:

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = 'could not convert non-number to float'
    return retval
```

The only thing we changed in the example was to return an error string as opposed to just `None`. We should take our function out for a "test drive" to see how well it works so far:

```
>>> safe_float('12.34') 12.34
>>> safe_float('bad input')
'could not convert non-number to float'
```

We made a good start—now we can detect invalid string input, but we are still vulnerable to invalid *objects* being passed in:


```
>>> safe_float({'a': 'Dict'}) Traceback (innermost last):
File "<stdin>", line 1, in ?
File "safeflt.py", line 28, in safe_float retval = float(object)
TypeError: object can't be converted to float
```

We will address this final shortcoming momentarily, but before we further modify our example, we would like to highlight the flexibility of the **try-except** syntax, especially the **except** statement, which comes in a few more flavors.

tryStatement with Multiple excepts

Earlier in this chapter, we introduced the following general syntax for **except**:

```
except Exception:
    suite_for_exception_Exception
```

The **except** statement in such formats specifically detects exceptions named *Exception*. You can chain multiple **except** statements together to handle different types of exceptions with the same **try**:

```
except Exception1:
    suite_for_exception_Exception1
except Exception2:
    suite_for_exception_Exception2
:
```

This same **try** clause is attempted, and if there is no error, execution continues, passing all the **except** clauses. However, if an exception *does* occur, the interpreter will look through your list of handlers attempting to match the exception with one of your handlers (**except** clauses). If one is found, execution proceeds to *that* **except**suite.

Our `safe_float()` function has some brains now to detect specific exceptions. Even smarter code would handle each appropriately. To do that, we have to have separate **except** statements, one for each exception type. That is no problem as Python allows **except** statements can be chained together. Any reader familiar with popular third-generation languages (3GLs) will no doubt notice the similarities to the switch/case statement which is absent in Python. We will now create separate messages for each error type, providing even more detail to the user as to the cause of his or her problem:

```
def safe_float(object):
    try:
        retval = float(object)
    except ValueError:
        retval = 'could not convert non-number to float'
    except TypeError:
        retval = 'object type cannot be converted to float'
    return retval
```


Running the code above with erroneous input, we get the following:

```
>>> safe_float('xyz')
'could not convert non-number to float'
>>> safe_float(()) 'argument must be a string'
>>> safe_float(200L) 200.0
>>> safe_float(45.67000) 45.67
```

exceptStatement with Multiple Exceptions

We can also use the same **except** clause to handle multiple exceptions. **except** statements which process more than one exception require that the set of exceptions be contained in a tuple:

```
except (Exception1, Exception2): suite_for_Exception1_and_Exception2
```

The above syntax example illustrates how two exceptions can be handled by the same code. In general, any number of exceptions can follow an **except** statement as long as they are all properly enclosed in a tuple:

```
except (Exception1[, Exception2[, ... ExceptionN...]]):
suite_for_exceptions_Exception1_to_ExceptionN
```

If for some reason, perhaps due to memory constraints or dictated as part of the design that all exceptions for our `safe_float()` function must be handled by the same code, we can now accommodate that requirement:

```
def safe_float(object):
try:
    retval = float(object)
except (ValueError, TypeError):
    retval = 'argument must be a number or numeric string'
return retval
```

Now there is only the single error string returned on erroneous input:

```
>>> safe_float('Spanish Inquisition') 'argument must be a number or numeric string'
>>> safe_float([])
'argument must be a number or numeric string'
>>> safe_float('1.6') 1.6
>>> safe_float(1.6) 1.6
>>> safe_float(932) 932.0
```

try-exceptwith No Exceptions Named

The final syntax for **try-except** we are going to present is one which does not specify an exception on the **except** header line:

```
try:
    try_suite           # watch for exceptions here
except:
    except_suite       # handles all exceptions
```

Although this code "catches the most exceptions," it does not promote good Python coding style. One of the chief reasons is that it does not take into account the potential root causes of problems which may generate exceptions. Rather than investigating and discovering what types of errors may occur

and how they may be prevented from happening, this type of code "turns the blind eye," thereby ignoring the possible causes (and remedies).

"Exceptional Arguments"

we are referring to the fact that exception may have *arguments* are passed along to the exception handler when they are raised. When an exception is raised, parameters are generally provided as an additional aid for the exception handler. Although arguments to exceptions are optional, the standard built-in exceptions do provide at least one argument, an error string indicating the cause of the exception.

Exception parameters can be ignored in the handler, but the Python provides syntax for saving this value. To access any provided exception argument, you must reserve a variable to hold the argument. This argument is given on the **except** header line and follows the exception type you are handling. The different syntaxes for the **except** statement can be extended to the following:

```
# single exception
except Exception, Argument:
    suite_for_Exception_with_Argument
# multiple exceptions
except (Exception1, Exception2, ..., ExceptionN), Argument:
    suite_for_Exception1_to_ExceptionN_with_Argument
```

Unless a string exception was raised, *Argument* is a class instance containing diagnostic information from the code raising the exception. The exception arguments themselves go into a tuple which is stored as an attribute of the class instance, an instance of the exception class from which it was instantiated. In the first alternate syntax above, *Argument* would be an instance of the `Exceptionclass`. For most standard built-in exceptions, that is, exceptions derived from `StandardError`, the tuple consists of a single string indicating the cause of the error. The actual exception name serves as a satisfactory clue, but the error string enhances the meaning even more. Operating system or other environment type errors, i.e., `IOError`, will also include an operating system error number which precedes the error string in the tuple. Whether an *Argument* is merely a string or a combination of an error number and a string, calling `str(Argument)` should present a human-readable cause of an error. The only caveat is that not all exceptions raised in third-party or otherwise external modules adhere to this standard protocol (or error string or (error number, error string)). We recommend to follow such a standard when raising your own exceptions (see Core Style note).

NOTE

When you raise built-in exceptions in your own code, try to follow the protocol established by the existing Python code as far as the error information that is part of the tuple passed as the exception argument. In other words, if you raise a `ValueError`, provide the same argument information as when the interpreter raises a `ValueError` exception, and so on. This helps keep the code consistent and will prevent other code which uses your module from breaking.

The example below is when an invalid object is passed to the float() built-in function, resulting in a TypeError exception:

```
>>> try:
...     float(['float() does not', 'like lists', 2])
...     except TypeError, diag:# capture diagnostic info
...     pass
...
>>> type(diag)
<type 'instance'>
>>>
>>> print diag
object can't be converted to float
```

The first thing we did was cause an exception to be raised from within the **try** statement. Then we passed cleanly through by ignoring but saving the error information. Calling the type() built-in function, we were able to confirm that our exception was indeed an instance. Finally, we displayed the error by calling print with our diagnostic exception argument.

To obtain more information regarding the exception, we can use the special class instance attribute which identifies which class an instance was instantiated from. Class objects also have attributes, such as a documentation string and a string name which further illuminate the error type:

```
>>> diag                                # exception instance object
<exceptions.TypeError instance at 8121378>
>>> diag._class_                         # exception class object
<class exceptions.TypeError at 80f6d50>
>>> diag._class_._doc_                   # exception class documentation string
'Inappropriate argument type.'
>>> diag._class_._name 'TypeError'      # exception class name
```

We will now update our safe_float() one more time to include the exception argument which is passed from the interpreter from within float() when exceptions are generated. In our last modification to safe_float(), we merged both the handlers for the ValueError and TypeError exceptions into one because we had to satisfy some requirement. The problem, if any, with this solution is that no clue is given as to which exception was raised nor what caused the error. The only thing returned is an error string which indicated some form of invalid argument. Now that we have the exception argument, this no longer has to be the case.

Because each exception will generate its own exception argument, if we chose to return this string rather than a generic one we made up, it would provide a better clue as to the source of the problem. In the following code snippet, we replace our single error string with the string representation of the exception argument.

```
def safe_float(object):
    try:
        retval = float(object)
    except (ValueError, TypeError), diag:
        retval = str(diag)
    return retval
```

Upon running our new code, we obtain the following (different) messages when providing improper input to `safe_float()`, even if both exceptions are managed by the same handler:

```
>>> safe_float('xyz')
'invalid literal for float(): xyz'
>>> safe_float({})
'object can't be converted to float'
```

Using Our Wrapped Function in an Application

We will now feature `safe_float()` in a mini application which takes a credit card transaction data file (`carddata.txt`) and reads in all transactions, including explanatory strings. Here are the contents of our example `carddata.txt` file:

```
% cat carddata.txt # carddata.txt previous balance 25
debits 21.64
541.24
25
credits
-25
-541.24
finance charge/late fees 7.30
5
```

Our program, `cardrun.py`, is given in [Example 10.1](#).

Example 10.1. Credit Card Transactions (`cardrun.py`)

We use `safe_float()` to process a set of credit card transactions given in a file and read in as strings. A log file tracks the processing.


```

<$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import types
004 4
005 5  def safe_float(object):
006 6      'safe version of float()'
007 7      try: <$nopage>
008 8          retval = float(object)
009 9      except (ValueError, TypeError), diag:
010 10         retval = str(diag)
011 11         return retval
012 12
013 13 def main():
014 14     'handles all the data processing'
015 15     log = open('cardlog.txt', 'w')
016 16     try: <$nopage>
017 17         ccfile = open('carddata.txt', 'r')
018 18     except IOError:
019 19         log.write('no txns this month\n')
020 20         log.close()
021 21         return <$nopage>
022 22
023 23     txns = ccfile.readlines()
024 24     ccfile.close()
025 25     total = 0.00
026 26     log.write('account log:\n')
027 27
028 28     for eachTxn in txns:
029 29         result = safe_float(eachTxn)
030 30         if type(result) == types.FloatType:
031 31             total = total + result
032 32             log.write('data... processed\n')
033 33         else: <$nopage>
034 34             log.write('ignored: %s' % result)
035 35     print '$%.2f (new balance)' % (total)
036 36     log.close()
037 37
038 38 if __name__ == '__main__':
039 39     main()
040 <$nopage>

```

Lines 1 – 3

The script starts by importing the types modules, which contains Type objects for the Python types. That is why we direct them to standard error instead.

Lines 5 – 11

This chunk of code contains the body of our safe_float()function.

Lines 13 – 36

The core part of our application performs three major tasks: (1) read the credit card data file, (2) process the input, and (3) display the result. Lines 16–24 perform the extraction of data from the file. You will notice that there is a **try-except** statement surrounding the file open.

A log file of the processing is also kept. In our example, we are assuming the log file can be opened for write without any problems. You will find that our progress is kept by the log. If the credit card data file cannot be accessed, we will assume there are no transactions for the month (lines 18–21).

The data is then read into the txns (transactions) list where it is iterated over in lines 28–34. After every call to `safe_float()`, we check the result type using the `types` module. The `types` module contains items of each type, named appropriately `typeType`, so that direct comparisons can be performed with results that determine an object's type. In our example, we check to see if `safe_float()` returns a string or float. Any string indicates an error situation with a string that could not be converted to a number, while all other values are floats which can be added to the running subtotal. The final new balance is then displayed as the final line of the `main()` function.

Lines 38 – 39

These lines represent the general "start only if not imported" functionality. Upon running our program, we get the following output:

```
% cardrun.py
$58.94 (new balance)
```

Taking a peek at the resulting log file (`cardlog.txt`), we see that it contains the following log entries after `cardrun.py` processed the transactions found in `carddata.txt`:

```
% cat cardlog.txt account log:
ignored: invalid literal for float(): # carddata.txt ignored: invalid literal for float():
previous balance data... processed
ignored: invalid literal for float(): debits data... processed
data... processed data... processed
ignored: invalid literal for float(): credits data... processed
data... processed
ignored: invalid literal for float(): finance charge/ late fees
data... processed data... processed
```

elseClause

We have seen the `else` statement with other Python constructs such as conditionals and loops.

With respect to `try-except`

statements, its functionality is not that much different from anything else you have seen: The `else` clause executes if no exceptions were detected in the preceding `try` suite.

All code within the `try` suite must have completed successfully (i.e., concluded with no exceptions raised) before any code in the `else` suite begins execution. Here is a short example in Python pseudocode:

```
import 3rd_party_module
log = open('logfile.txt', 'w')
try:
    3rd_party_module.function()
except:
    log.write("*** caught exception in module\n")
else:
    log.write("*** no exceptions caught\n") log.close()
```

In the above example, we import an external module and test it for errors. A log file is used to

determine whether there were defects in the third-party module code. Depending on whether an exception occurred during execution of the external function, we write differing messages to the log.

try-exceptKitchen Sink

We can combine all the varying syntaxes that we have seen so far in this chapter to highlight all the different ways you can use

try-except-else

```
try:
    try_suite
except Exception1:
    suite_for_Exception1
except (Exception2, Exception3, Exception4) :
    suite_for_Exceptions_2_3_and_4
except Exception5, Argument5:
    suite_for_Exception5_plus_argument
except (Exception6, Exception7) , Argument67:
    suite_for_Exceptions6_and_7_plus_argument
except:
    suite_for_all_other_exceptions
else:
    no_exceptions_detected_suite
```

try-finallyStatement

The **try-finally** statement differs from its **try-except** brethren in that it is not used to handle exceptions. Instead it is used to maintain consistent behavior regardless of whether or not exceptions occur. The **finally** suite executes regardless of an exception being triggered within the **try** suite.

```
try:
    try_suite
finally:
    finally_suite # executes regardless of exceptions
```

When an exception does occur within the **try** suite, execution jumps immediately to the **finally** suite. When all the code in the **finally** suite completes, the exception is re-raised for handling at the next higher layer. Thus it is common to see a **try-finally** nested as part of a **try-except** suite.

One place where we can add a **try-finally** statement is by improving our code in `cardrun.py` so that we catch any problems which may arise from reading the data from the `carddata.txt` file. In the current code in [Example 10.1](#), we do not detect errors during the read phase (using `readlines()`):

```
try:
    ccfile = open('carddata.txt')
except IOError:
    log.write('no txns this month\n') log.close()
return
txns = ccfile.readlines() ccfile.close()
```

It is possible for `readlines()` to fail for any number of reasons, one of which is if `carddata.txt` was a file on the network (or a floppy) that became inaccessible. Regardless, we should improve this piece of code so that the entire input of data is enclosed in the **try** clause:

```

try:
    ccfile = open('carddata.txt') txns = ccfile.readlines() ccfile.close()
except IOError:
    log.write('no txns this month\n') log.close()
return

```

All we did was to move the `readlines()` and `close()` method calls to the `try` suite. Although our code is more robust now, there is still room for improvement. Notice what happens if there was an error of some sort. If the `open` succeeds but for some reason the `readlines()` call does not, the exception will continue with the **except** clause. No attempt is made to close the file. Wouldn't it be nice if we closed the file regardless of whether an error occurred or not? We can make it a reality using **try-finally**:

```

try:
    ccfile = open('carddata.txt')
try:
    txns = ccfile.readlines()
finally:
    ccfile.close()
except IOError:
    log.write('no txns this month\n') log.close()
return

```

Now our code is more robust than ever. Let us take a look at another familiar example, calling `float()` with an invalid value. We will use **print** statements to show you the flow of execution within the **try-except** and **try-finally** clauses. We present `tryfin.py` in [Example 10.2](#).

Example 10.2. Testing the try-finallyStatement (tryfin.py)

This small script simply illustrates the flow of control when using a try-finally statement embedded within the try clause of a try-exceptstatement.

```

<$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  try: <$nopage>
004 4      print 'entering 1st try'
005 5      try: <$nopage>
006 6          print 'entering 2nd try'
007 7          float('abc')
008 8
009 9      finally: <$nopage>
010 10         print 'doing finally'
011 11
012 12  except ValueError:
013 13     print 'handling ValueError'
014 14
015 15  print 'finishing execution'
016 <$nopage>

```

Running this code, we get the following output:

```
% tryfin.py
entering 1st try
entering 2nd try
doing finallyhandling ValueError
finishing execution
```

One final note: If the code in the **finally** suite raises another exception, or is aborted due to a **return**,**break**, or **continue** statement, the original exception is lost and cannot be re-raised. Quick review: The **try-finally** statement presents a way to detect errors but ignore other than cleanup, and passes the exception up to higher layers for possible handling.

11. Explain in detail about Exception as strings.

*Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have any relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

For backwards compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option `-X` will provide you with the standard exceptions as strings. This feature will be obsoleted beginning with Python 1.6.

If you must use string exceptions, we will now show you how to do it right. The following piece of code may or may not work:

```
# this may not work... risky!
try:
:
raise 'myexception'
:
except 'myexception'
suite_to_handle_my_string_exception
except:
suite_for_other_exceptions
```

The reason why the above code may not work is because exceptions are based on object identity as opposed to object value. There are two different string objects above, both with the same value. To rectify the potential problem, create a static string object with which to use:

```
# this is a little bit better myexception = 'myexception'
try:
:
raise myexception
:
except myexception:
suite_to_handle_my_string_exception
except:
suite_for_other_exceptions
```

With this update, the same string object is used. However, if you are going to use this code, you might

as well use an exception class. Substitute the myexception assignment above with:

```
# this is the best choice
class MyException(Exception):
    pass

    :
try:
    :
    raise MyException
    :
except MyException:
    suite_to_handle_my_string_exception
except:
    suite_for_other_exceptions
```

So you see, there really is no reason *not* to use exception classes from now on when creating your own exceptions. Be careful, however, because you may end up using an external module which may still have exceptions implemented as strings.

*Exceptions as Classes

As we mentioned above, as of Python 1.5, all standard exceptions are now identified using classes. User-defined, class-based exceptions have been around for longer than that (since Python 1.2!), but until 1.5, the standard exceptions remained implemented as strings, mostly for backwards compatibility. However, there are a number of advantages that classes bring to the table, and these reasons were what finally led to all standard exceptions being converted from strings to class-based.

Selection via Object Identity

The search for an exception handler (checking each **except** clause) is accomplished via object identity and not object value. That means that if you are using string exceptions, the string object used in the **except** clause must be the same as the string exception that is raised. Two different string objects, even if they contain exactly the same string, constitute different exceptions!

Using classes simplifies this selection mechanism because exception classes are, for the most part, static. When referring to an exception, you are really accessing a class object that is a built-in identifier and stays constant throughout the course of execution. Whether using `IndexError` in an **except** clause or in a **raise** statement, you can be sure that they are both referencing the same class object so that a corresponding handler will be found.

Relationship Between Exceptions

Utilizing classes also allows for a hierarchical structure of exceptions. There are two consequences of employing this construct:

Promotes Grouping of Related Exceptions

When errors were simply strings, there was no interrelationship between any pair of errors. Although most errors are unrelated, some *are* very closely related, such as `IndexError`—offset into a sequence with an invalid index, and `KeyError`—indexing into a map with an invalid key. String exceptions allow these exceptions to be related in context or description only and do not recognize any more than that codewise.

Class-based exceptions allow such a relationship. Both exceptions now are subclassed from a common ancestor, the `LookupError` exception. If your application defined a new class with a lookup-related error, it is now possible for you to create yet another related exception simply by also subclassing from `LookupError`, or even `IndexError` or `KeyError`.

Simplifies Detection

With class-based exceptions, handler code can detect an entire exception class "tree" (i.e., an ancestor exception class as well as all derived subclasses). As an example, let us say that you just want to catch any general arithmetic error in your program. Our code may be structured something like the following:

```
try:
    code_to_scan_for_math_errors
except FloatingPointError:
    print "math exception found"
except ZeroDivisionError:
    print "math exception found"
except OverflowError:
    print "math exception found"
```

Since the handlers for each exception are the same, we can shorten the code to:

```
try:
    code_to_scan_for_math_errors
except (FloatingPointError, ZeroDivisionError, OverflowError):
    print "math exception found"
```

However, this solution is not as all-encompassing as it could be, is a little messy perhaps with all three exceptions listed, and does not take into account future expansion. What if the next version of Python comes with a new arithmetic exception, or perhaps you create such a new exception for your application? The code we have above would be out-of-date and inaccurate.

The solution is to reference a base class in your **except** clause. Because your new exceptions (as well as `FloatingPointError`, `ZeroDivisionError`, and `OverflowError`) are all subclassed from the `ArithmeticError` exception class, you can reference `ArithmeticError` which can then scan for all `ArithmeticError` exceptions as well as all exceptions *derived* from `ArithmeticError`. Updating our code one more time, we present the most flexible solution here:

```
try:
    code_to_scan_for_math_errors
except ArithmeticError:
    print "math exception found"
```

Now your code can handle all pre-existing `ArithmeticError` exceptions as well as any you may create subclassed from `ArithmeticError`. Care must be taken, however, when handling both classes and

superclasses with the same **try** statement. Observe both of the following examples:

```
try:  
code_to_scan_for_math_errors  
except ArithmeticError:  
print "math exception found"  
except ZeroDivisionError:  
print "division by zero error"
```

```
try:  
code_to_scan_for_math_errors  
except ZeroDivisionError:  
print "division by zero error"  
except ArithmeticError:  
print "math exception found"
```

Exception handlers are mutually-exclusive, meaning that once a handler is found for an exception (or a base class), it is handled immediately without searching further. In the first example, a `ZeroDivisionError` will be handled only by the first **except** statement, producing an output of "math exception found." The **except** clause for `ZeroDivisionError` will not be reached.

The second example may prove to be more useful, as a specific arithmetic error (`ZeroDivisionError`) is handled first, leaving the general `ArithmeticError` handler to take care of any other exception derived from `ArithmeticError`.

12. Explain about Raising exceptions with example.

Raising Exceptions

The interpreter was responsible for raising all of the exceptions which we have seen so far. These exist as a result of encountering an error during execution. A programmer writing an API may also wish to throw an exception on erroneous input, for example, so Python provides a mechanism for the programmer to explicitly generate an exception: the **raise** statement.

raise Statement

The **raise** statement is quite flexible with the arguments which it supports, translating to a large number of different formats supported syntactically. The general syntax for **raise** is:

```
raise [Exception [, args [, traceback]]]
```

The first argument, `Exception`, is the name of the exception to raise. If present, it must either be a string, class, or instance (more below). `Exception` must be given if any of the other arguments (arguments or `traceback`) are present. A list of all Python standard exceptions is given in [Table 10.2](#).

The second expression contains optional *args* (a.k.a. parameters, values) for the exception. This value is either a single object or a tuple of objects. When exceptions are detected, the exception arguments are always returned as a tuple. If *args* is a tuple, then that tuple represents the same set of exception arguments which are given to the handler. If *args* is a single object, then the tuple will consist solely of this one object (i.e., a tuple with one element). In most cases, the single argument consists of a string indicating the cause of the error. When a tuple is given, it usually equates to an error string, an error number, and perhaps an error location, such as a file, etc.

The final argument, *traceback*, is also optional (and rarely used in practice), and, if present, is the traceback object used for the exception—normally a traceback object is newly created when an exception is raised. This third argument is useful if you want to re-raise an exception (perhaps to point to the previous location from the current). Arguments which are absent are represented by the value **None**.

The most common syntax used is when *Exception* is a class. No additional parameters are ever required, but in this case, if they are given, can be a single object argument, a tuple of arguments, or an exception class instance. If the argument is an instance, then it can be an instance of the given class or a derived class (subclassed from a pre-existing exception class). No additional arguments (i.e., exception arguments) are permitted if the argument is an instance.

What happens if the argument is an instance? No problems arise if instance is an instance of the given exception class. However, if instance is *not* an instance of the class nor an instance of a subclass of the class, then a new instance of the exception class will be created with exception arguments copied from the given instance. If instance is an instance of a subclass of the exception class, then the new exception will be instantiated from the subclass, not the original exception class.

If the additional parameter to the **raise** statement used with an exception class is not an instance—instead, it is a singleton or tuple—then the class is instantiated and *args* is used as the argument list to the exception. If the second parameter is not present or None, then the argument list is empty.

If *Exception* is an instance, then we do not need to instantiate anything. In this case, additional parameters must not be given or must be None. The exception type is the class which *instance* belongs to; in other words, this is equivalent to raising the class with this instance, i.e., **raise instance._class_, instance**.

Use of string exceptions is deprecated in favor of exception classes, but if *Exception* is a string, then it raises the exception identified by *string*, with any optional parameters (*args*) as arguments.

Finally, the **raise** statement by itself without any parameters is a new construct, introduced in Python 1.5, and causes the last exception raised in the current code block to be re-raised. If no exception was previously raised, a `TypeError` exception will occur, because there was no previous exception to re-raise.

raise syntax	Description
raise <i>exclass</i>	raise an exception, creating an instance of <i>exclass</i> (without any exception arguments)
raise <i>exclass</i> ()	same as above since classes are now exceptions; invoking the classname with the function call operator instantiates an instance of <i>exclass</i> , also with no arguments
raise <i>exclass, args</i>	same as above, but also providing exception arguments <i>args</i> , which can be a single argument or a tuple
raise <i>exclass</i> (<i>args</i>)	same as above
raise <i>exclass, args, tb</i>	same as above, but provides traceback object <i>tb</i> to use
raise <i>exclass, instance</i>	raise exception using <i>instance</i> (normally an instance of <i>exclass</i>); if <i>instance</i> is an instance of a subclass of <i>exclass</i> , then the new exception will be of the subclass type (not of <i>exclass</i> type); if <i>instance</i> is <i>not</i> an instance of <i>exclass</i> <i>nor</i> an instance of a subclass of <i>exclass</i> , then a new instance of <i>exclass</i> will be created with exception arguments copied from <i>instance</i>
raise <i>instance</i>	raise exception using <i>instance</i> : the exception type is the class which instantiated <i>instance</i> ; equivalent to raise <i>instance._class_, instance</i> (same as above)
raise <i>string</i>	(<i>archaic</i>) raises <i>string</i> exception
raise <i>string, args</i>	same as above, but raises exception with <i>args</i>
raise <i>string, args, tb</i>	same as above, but provides traceback object <i>tb</i> to use
raise	(<i>new in 1.5</i>) re-raises previously raised exception; if no exception was previously raised, a <code>TypeError</code> is raised

Table 10.1. Using the raiseStatement

13. What is meant by an Assertions?

Assertions

Assertions are diagnostic predicates which must evaluate to Boolean true; otherwise, an exception is raised to indicate that the expression is false. These work similarly to the assert macros which are part of the C language preprocessor, but in Python these are run-time constructs (as opposed to pre-compile directives).

If you are new to the concept of assertions, no problem. The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the **assert** statement, the newest keyword to Python, introduced in version 1.5.

assertStatement

The **assert** statement evaluates a Python expression, taking no action if the assertion succeeds (similar to a **pass** statement), but otherwise raises an `AssertionError` exception. The syntax for **assert** is:

```
assert expression [, arguments]
```

Here are some examples of the use of the **assert** statement:

```
assert 1 == 1
```

```
assert (2 + 2) == (2 * 2)
```

```
assert len(['my list', 12]) < 10
```



```
assert range(3) == [0, 1, 2]
```

AssertionError exceptions can be caught and handled like any other exception using the **try-except** statement, but if not handled, they will terminate the program and produce a traceback similar to the following:

```
>>> assert 1 == 0
Traceback (innermost last): File "<stdin>", line 1, in?
AssertionError
```

Like the **raise** statement we investigated in the previous section, we can provide an exception argument to our **assert** command:

```
>>> assert 1 == 0, 'One does not equal zerosilly!'
Traceback (innermost last):
File "<stdin>", line 1, in ?
AssertionError: One does not equal zero silly!
```

Here is how we would use a **try-except** statement to catch an AssertionError exception:

```
try:
    assert 1 == 0, 'One does not equal zero silly!'
except AssertionError, args:
    print '%s: %s' % (args._class._name, args)
```

Executing the above code from the command-line would result in the following output:

```
AssertionError: One does not equal zero silly!
```

To give you a better idea of how **assert** works, imagine how the **assert** statement may be implemented in Python if written as a function. It would probably look something like this:

```
def assert(expr, args=None):
    if debug and not expr:
        raise AssertionError, args
```

The first **if** statement confirms the appropriate syntax for the **assert**, meaning that **expr** should be an expression. We compare the type of **expr** to a real expression to verify. The second part of the function evaluates the expression and raises **AssertionError**, if necessary. The built-in variable **debug** is 1 under normal circumstances, 0 when optimization is requested (command line option -O).

14. Explain about standard exceptions.

Standard Exceptions

[Table 10.2](#) lists all of Python's current set of standard exceptions. All exceptions are loaded into the interpreter as a built-in so they are ready before your script starts or by the time you receive the interpreter prompt, if running interactively.

All standard/built-in exceptions are derived from the root class **Exception**. There are currently two immediate subclasses of **Exception**: **SystemExit** and **StandardError**. All other built-in exceptions are subclasses of **StandardError**. Every level of indentation of an exception listed in [Table 10.2](#) indicates one level of exception class derivation.

Table 10.2. Python Standard Exceptions

<i>Exception Name</i>	<i>Description</i>
Exception ^[a]	root class for all exceptions
SystemExit	request termination of Python interpreter
StandardError ^[a]	base class for all standard built-in exceptions
ArithmeticError ^[a]	base class for all numeric calculation errors
FloatingPointError ^[a]	error in floating point calculation
OverflowError	calculation exceeded maximum limit for numerical type
ZeroDivisionError	division (or modulus) by zero error (all numeric types)
AssertionError ^[a]	failure of <code>assert</code> statement
AttributeError	no such object attribute
EOFError	end-of-file marker reached without input from built-in
EnvironmentError ^[b]	base class for operating system environment errors
IOError	failure of input/output operation
OSError ^[b]	operating system error
WindowsError ^[c]	MS Windows system call failure
ImportError	failure to import module or object
KeyboardInterrupt	user interrupted execution (usually by typing ^C)
LookupError ^[a]	base class for invalid data lookup errors
IndexError	no such index in sequence
KeyError	no such key in mapping
MemoryError	out-of-memory error (non-fatal to Python interpreter)
NameError	undeclared/uninitialized object (non-attribute)
UnboundLocalError ^[c]	access of an uninitialized local variable
RuntimeError	generic default error during execution
NotImplementedError ^[b]	unimplemented method
SyntaxError	error in Python syntax
IndentationError ^[d]	improper indentation
TableError ^[d]	improper mixture of TABs and spaces
SystemError	generic interpreter system error
TypeError	invalid operation for type
ValueError	invalid argument given
UnicodeError ^[c]	Unicode related error

[a] Prior to Python 1.5, the exceptions denoted did not exist. All earlier exceptions were string-based.

[b] New as of Python 1.5.2

[c] New as of Python 1.6

[d] New as of Python 2.0

15. Show how to create an exception?

*Creating Exceptions

Although the set of standard exceptions is fairly wide-ranging, it may be advantageous to create your own exceptions. One situation is where you would like additional information from what a standard or module-specific exception provides. We will present two examples, both related to IOError. IOError is a generic exception used for input/output problems which may arise from invalid

file access or other forms of communication. Suppose we wanted to be more specific in terms of identifying the source of the problem. For example, for file errors, we want to have a FileError exception which behaves like IOError, but with a name that has more meaning when performing file operations.

Another exception we will look at is related to network programming with sockets. The exception generated by the socket module is called socket.error and is not a built-in exception. It is subclassed from the generic Exception exception. However, the exception arguments from socket.error closely resemble those of IOError exceptions, so we are going to define a new exception called NetworkError which subclasses from IOError but contains at least the information provided by socket.error.

We now present a module called myexc.py with our newly-customized exceptions

FileError and NetworkError. The code is in [Example 10.3](#)

Example 10.3 Creating Exceptions (myexc.py)

This module defines two new exceptions, FileError and NetworkError, as well as reimplements more diagnostic versions of open() [myopen()] and socket.connect() [myconnect()]. Also included is a test function [test()] that is run if this module is executed directly.

<\$nopcode>

```
001 1      #!/usr/bin/env python
002 2
003 3      import      os, socket, errno, types, tempfile 004 4
005 5      class NetworkError(IOError):
006 6          pass <$nopcode>
007 7
008 8      class FileError(IOError):
009 9          pass <$nopcode>
010 10
011 11      def updArgs(args, newarg=None):
012 12
013 13          if      type(args) == types.InstanceType:
014 14              myargs = []
015 15              for eachArg in args:
016 16                  myargs.append(eachArg)
017 17          else: <$nopcode>
018 18              myargs = list(args)
019 19
020 20          if newarg:
021 21              myargs.append(newarg)
022 22
023 23          return tuple(myargs)
024 24
025 25      def fileArgs(file, mode, args):
026 26
027 27          if      args[0] == errno.EACCES and \
028 28              'access' in dir(os):
029 29              perms = "
```

```

030 30         permd = { 'r': os.R_OK, 'w': os.W_OK,
031 31             'x': os.X_OK}
032 32         pkeys = permd.keys()
033 33         pkeys.sort()
034 34         pkeys.reverse()
035 35
036 36         for eachPerm in 'rwx':
037 37             if os.access(file, permd[eachPerm]):
038 38                 perms = perms + eachPerm
039 39             else: <$nopage>
040 40                 perms = perms + '-'
041 41
042 42             if type(args) == types.InstanceType:
043 43                 myargs = []
044 44             for eachArg in args:
045 45                 myargs.append(eachArg)
046 46             else: <$nopage>
047 47                 myargs = list(args)
048 48
049 49             myargs[1] = "'%s' %s (perms: '%s')" % \
050 50             (mode, myargs[1], perms)
051 51
052 52             myargs.append(args.filename)
053 53
054 54             else: <$nopage>
055 55                 myargs = args
056 56
057 57             return tuple(myargs)
058 58
059 59 def myconnect(sock, host, port):
060 60
061 61     try: <$nopage>
062 62         sock.connect((host, port))
063 63
064 64     except socket.error, args:
065 65         myargs = updArgs(args)# conv inst2tuple 066 66         if len(myargs) ==
066 66         1:# no #s on some errs 067 67         myargs = (errno.ENXIO, myargs[0])
068 68
069 69         raise NetworkError, \
070 70         updArgs(myargs, host + ':' + str(port)) 071 71
072 72 def myopen(file, mode='r'):
073 73
074 74     try: <$nopage>
075 75         fo = open(file, mode)

```

```

076 76
077 77 except IOError, args:
078 78 raise FileError, fileArgs(file, mode, args)
079 79
080 80 return fo
081 81
082 82 def testfile():
083 83
084 84 file = mktemp()
085 85 f = open(file, 'w')
086 86 f.close()
087 87
088 88 for eachTest in ((0, 'r'), (0100, 'r'), \
089 89 0400, 'w'), (0500, 'w')):
090 90 try: <$nopcode>
091 91 os.chmod(file, eachTest[0])
092 92 f = myopen(file, eachTest[1])
093 93
094 94 except FileError, args:
095 95 print "%s: %s" % \
096 96 (args._class_.name_, args)
097 97 else: <$nopcode>
098 98 print file, "opened ok... perm ignored"
099 99 f.close()
100 100
101 101 os.chmod(file, 0777)# enable all perms
102 102 os.unlink(file)
103 103
104 104 def testnet():
105 105 s = socket.socket(socket.AF_INET, \
106 106 socket.SOCK_STREAM)
107 107
108 108 for eachHost in ('deli', 'www'):
109 109 try: <$nopcode>
110 110 myconnect(s, 'deli', 8080)
111 111 except NetworkError, args:
112 112 print "%s: %s" % \
113 113 (args._class_.name_, args)
114 114
115 115 if _name == '_main_':
116 116     testfile()
117 117     testnet()
118     <$nopcode>

```


Lines 1 – 3

The Unix start-up script and importation of the socket, os, errno, types, and temp file modules help us start this module.

Lines 5 – 9

Believe it or not, these five lines make up our new exceptions. Not just one, but both of them. Unless new functionality is going to be introduced, creating a new exception is just a matter of subclassing from an already-existing exception. In our case, that would be IOError. EnvironmentError, from which IOError is derived would also work, but we wanted to convey that our exceptions were definitely I/O-related.

We chose IOError because it provides two arguments, an error number and an error string. File-related [uses open()] IOError exceptions even support a third argument which is not part of the main set of exception arguments, and that would be the file name. Special handling is done for this third argument which lives outside the main tuple pair and has the name filename.

Lines 11 – 23

The entire purpose of the updArgs() function is to "update" the exception arguments. What we mean here is that the original exception is going to provide us a set of arguments. We want to take these arguments and make them part of our new exception, perhaps embellishing or adding a third argument (which is not added if nothing is given— None is a default argument which we will study in the next chapter). Our goal is to provide the more informative details to the user so that if and when errors occur, the problems can be tracked down as quickly as possible.

Lines 25 – 57

The fileArgs() function is used only by myopen() [see below]. In particular, we are seeking error EACCES, which represents "permission denied." We pass all other IOError exceptions along without modification (lines 54–55). If you are curious about ENXIO, EACCES, and other system error numbers, you can hunt them down by starting at file /usr/include/sys/errno.h on a Unix system, or C:\Msdev\include\Errno.h if you are using Visual C++ on Windows.

In line 27, we are also checking to make sure that the machine we are using supports the os.access() function, which helps you check what kind of file permissions you have for any particular file. We do not proceed unless we receive both a permission error as well as the ability to check what kind of permissions we have. If all checks out, we set up a dictionary to help us build a string indicating the permissions we have on our file.

The Unix file system uses explicit file permissions for the user, group (more than one user can belong to a "group"), and other (any user other than the owner or someone in the same group as the owner) in read, write, and execute ('r', 'w', 'x') order. Windows supports some of these permissions.

Now it is time to build the permission string. If the file has a permission, its corresponding letter shows up in the string, otherwise a dash (-) appears. For example, a string of "rw-" means that you have read and write access to it. If the string reads "r-x", you have only read and execute access; "---" means no permission at all.

After the permission string has been constructed, we create a temporary argument list. We then alter the error string to contain the permission string, something which standard IOError exception does not provide. "Permission denied" sometimes seems silly if the system does not tell you what permissions you have to correct the problem. The reason, of course, is security. When intruders do not have

permission to access something, the last thing you want them to see is what the file permissions are, hence the dilemma. However, our example here is merely an exercise, so we allow for the temporary "breach of security." The point is to verify whether or not the `os.chmod()` functions call affected file permissions the way they are supposed to. The final thing we do is to add the file name to our argument list and return the set of arguments as a tuple.

Lines 59 – 70

Our new `myconnect()` function simply wraps the standard `socket` method `connect()` to provide an `IOError`-type exception if the network connection fails. Unlike the general `socket.error` exception, we also provide the host name and port number as an added value to the programmer.

For those new to network programming, a host name and port number pair are analogous to an area code and telephone number when you are trying to contact someone. In this case, we are trying to contact a program running on the remote host, presumably a server of some sort; therefore, we require the host's name and the port number that the server is listening on.

When a failure occurs, the error number and error string are quite helpful, but it would be even more helpful to have the exact host-port combination as well, since this pair may be dynamically-generated or retrieved from some database or name service. That is the value-add we are bestowing to our version of `connect()`. Another issue arises when a

host cannot be found. There is no direct error number given to us by the `socket.error` exception, so to make it conform to the `IOError` protocol of providing an error number-error string pair, we find the closest error number that matches. We choose `ENXIO`.

Lines 72 – 80

Like its sibling `myconnect()`, `myopen()` also wraps around an existing piece of code. Here, we have the `open()` function. Our handler catches only `IOError` exceptions. All others will pass through and on up to the next level (when no handler is found for them).

Once an **`IOError`** is caught, we raise our own error and customized arguments as returned from `fileArgs()`.

Lines 82 – 102

We shall perform the file testing first, here using the `testfile()` function. In order to begin, we need to create a test file that we can manipulate by changing its permissions to generate permission errors. The `tempfile` module contains code to create temporary file names or temporary files themselves. We just need the name for now and use our new `myopen()` function to create an empty file. Note that if an error occurred here, there would be no handler, and our program would terminate fatally—the test program should not continue if we cannot even *create* a test file.

Our test uses four different permission configurations. A zero means no permissions at all, `0100` means execute-only, `0400` indicates read-only, and `0500` means read- and execute-only (`0400 + 0100`). In all cases, we will attempt to open a file with an invalid mode. The `os.chmod()` function is responsible for updating a file's permission modes. (NOTE: these permissions all have a leading zero in front, indicating that they are octal [base 8] numbers.)

If an error occurs, we want to display diagnostic information similar to the way the Python interpreter performs the same task when uncaught exceptions occur, and that is giving the exception name followed by its arguments. The `_class` special variable provides the class object for which an instance was created from. Rather than displaying the entire class name here (`myexc.FileError`), we use the class object's `_name` variable to just display the class name (`FileError`), which is also what you see from the interpreter in an unhandled error situation. Then the arguments which we arduously put together in

our wrapper functions follow.

If the file opened successfully, that means the permissions were ignored for some reason. We indicate this with a diagnostic message and close the file. Once all tests have been completed, we enable all permissions for the file and remove it with the `os.unlink()` function. `os.remove()` is equivalent to `os.unlink()`.

Lines 104 – 113

The next section of code (`testnet()`) tests our `NetworkError` exception. A socket is a communication endpoint with which to establish contact with another host. We create such an object, then use it in an attempt to connect to a host with no server to accept our connect request and a host not on our network.

Lines 115 – 117

We want to execute our `test*()` functions only when invoking this script directly, and that is what the code here does. Most of the scripts given in this text utilize the same format.

Running this on a Unix machine, we get the following output:

```
% myexc.py
FileError: [Errno 13] 'r' Permission denied (perms: '---'): '/usr/tmp/@18908.1'
FileError: [Errno 13] 'r' Permission denied (perms: '--x'): '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r--'): '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'): '/usr/tmp/@18908.1'
NetworkError: [Errno 146] Connection refused: 'deli:8080' NetworkError: [Errno 6] host not
found: 'www:8080'
```

The results are slightly different on a Windows machine:

```
D:\python>python myexc.py
C:\WINDOWS\TEMP\~-195619-1 opened ok... perms ignored C:\WINDOWS\TEMP\~-195619-1
opened ok... perms ignored
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'): 'C:\\WINDOWS\\TEMP\\~-195619-1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'): 'C:\\WINDOWS\\TEMP\\~-195619-1'
NetworkError: [Errno 10061] winsock error: 'deli:8080' NetworkError: [Errno 6] host not found:
'www:8080'
```

You will notice that Windows does not support read permissions on files, which is the reason why the first two file open attempts succeeded. Your mileage may vary (YMMV) on your own machine and operating system.

16. Why Exceptions (Now)?

There is no doubt that errors will be around as long as software is around. The difference in today's fast-paced computing world is that our execution environments have changed, and so has our need to adapt error-handling to accurately reflect the operating context of the software which we develop. Modern-day applications generally run as self-contained graphical user interfaces (GUIs) or in a client-server architecture such as the Web.

The ability to handle errors at the application level has become even more important recently in that users are no longer the only ones directly running applications. As the Internet and online electronic commerce become more pervasive, web servers will be the primary users of application software. This means that applications cannot just fail or crash outright anymore, because if they do, system errors translate to browser errors, and these in turn lead to frustrated users. Losing eyeballs means losing

advertising revenue and potentially significant amounts of irrecoverable business.

If errors do occur, they are generally attributed to some invalid user input. The execution environment must be robust enough to handle the application-level error and be able to produce a user-level error message. This must translate to a "non-error" as far as the web server is concerned because the application must complete successfully, even if all it does is return an error message to present to the user as a valid hypertext markup language (HTML) web page displaying the error.

If you are not familiar with what I am talking about, does a plain web browser screen with the big black words saying, "Internal Server Error" sound familiar? How about a fatal error that brings up a pop-up that declares "Document contains no data"? As a user, do either of these phrases mean anything to you? No, of course not (unless you are an Internet software engineer), and to the average user, they are an endless source of confusion and frustration. These errors are a result of a failure in the execution of an application. The application either returns invalid hypertext transfer protocol (HTTP) data or terminates fatally, resulting in the web server throwing its hands up into the air, saying, "I give up!" This type of faulty execution should not be allowed, if at all possible. As systems become more complex and involve more apprentice users, additional care should be taken to ensure a smooth user application experience. Even in the face of an error situation, an application should terminate successfully, as to not affect its execution environment in a catastrophic way. Python's exception handling promotes mature and correct programming.

17. Why Exceptions at All?

If the above section was not motivation enough, imagine what Python programming might be like without program-level exception handling. The first thing that comes to mind is the loss of control client programmers have over their code. For example, if you created an interactive application which allocates and utilizes a large number of resources, if a user hit ^C or other keyboard interrupt, the application would not have the opportunity to perform clean-up, resulting in perhaps loss of data, or data corruption. There is also no mechanism to take alternative action such as prompting the users to confirm whether they really want to quit or if they hit the control key accidentally.

Another drawback would be that functions would have to be rewritten to return a "special" value in the face of an error situation, for example, None. The engineer would be responsible for checking each and every return value from a function call. This may be cumbersome because you may have to check return values which may not be of the same type as the object you are expecting if no errors occurred. And what if your function wants to return None as a valid data value? Then you would have to come up with another return value, perhaps a negative number. We probably do not need to remind you that negative numbers may be valid in a Python context, such as an index into a sequence. As a programmer of application programmer interfaces (APIs), you would then have to document every single return error your users may encounter based on the input received. Also, it is difficult (and tedious) to propagate errors (and reasons) of multiple layers of code.

There is no simple propagation like the way exceptions do it. Because error data needs to be transmitted upwards in the call hierarchy, it is possible to misinterpret the errors along the way. A totally unrelated error may be stated as the cause when in fact it had nothing to do with the original problem to begin with. We lose the bottling-up and safekeeping of the original error that exceptions provide as they are passed from layer to layer, not to mention completely losing track of the data we were originally concerned about! Exceptions simplify not only the code, but the entire error management scheme which should not play such a significant role in application development. And

with Python's exception handling capabilities, it does not have to.

18. Explain about Exceptions and the sysModule.

Exceptions and the sysModule

An alternative way of obtaining exception information is by accessing the `exc_info()` function in the `sys` module. This function provides a 3-tuple of information, more than what we can achieve by simply using only the exception argument. Let us see what we get using `sys.exc_info()`:

```
>>> try:
...     float('abc123')
... except:
...     import sys
...     exc_tuple = sys.exc_info()
...
>>> print exc_tuple
(<class exceptions.ValueError at f9838>, <exceptions.ValueError instance at
122fa8>,
<traceback object at 10de18>)
>>>
>>> for eachItem in exc_tuple:
...     print eachItem
... exceptions.ValueError
invalid literal for float(): abc123
<traceback object at 10de18>
```

What we get from `sys.exc_info()` in a tuple are:

- exception class object
- (this) exception class instance object
- traceback object

The first two items we are familiar with: the actual exception class and this particular exception's instance (which is the same as the exception argument which we discussed in the previous section). The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred.

In older versions of Python, these three values were available in the `sys` module as `sys.exc_type`, `sys.exc_value`, and `sys.exc_traceback`. Unfortunately, these three are global variables and not thread-safe. We recommend using `sys.exc_info()` instead.

Related Modules

The classes found in module `Lib/exceptions.py` are automatically loaded as built-in names on start-up, so no explicit import of this module is ever necessary. We recommend you take a look at this source code to familiarize yourself with Python's exceptions and how they interrelate and interoperate. Starting with 2.0, exceptions are now built into the interpreter (see `Python/exceptions.c`).

Modules

19. What are Modules? Explain about Modules and Files.

A *module* allows you to logically organize your Python code. When code gets to be large enough, the tendency is to break it up into organized pieces which can still interact with each other at a functioning level. These pieces generally have attributes which have some relation to each other, perhaps a single class with its member data variables and methods, or maybe a group of related, yet independently operating functions. These pieces should be shared, so Python allows a module the ability to "bring in" and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating attributes from other modules with your module is called *importing*. Self-contained and organized pieces of Python code that can be shared in a nutshell, that describes a module.

Modules and Files

If modules represent a logical way to organize your Python code, then files are a way to physically organize modules. To that end, each file is considered an individual module, and vice versa. The file name of a module is the module name appended with the .py file extension. There are several aspects we need to discuss with regards to what the file structure means to modules. Unlike other languages in which you import classes, in Python you import modules or module attributes.

20. Describe about Namespaces in detail.

Namespaces

The basic concept of a namespace is an individual set of mappings from names to objects. As you are no doubt aware, module names play an important part in the naming of their attributes. The name of the attribute is always prepended with the module name. For example, the `atoi()` function in the `string` module is called `string.atoi()`. Because only one module with a given name can be loaded into the Python interpreter, there is no intersection of names from different modules; hence, each module defines its own unique namespace. If I created a function called `atoi()` in my own module, perhaps `mymodule`, its name would be `mymodule.atoi()`. So even if there is a name conflict for an attribute, the *fully-qualified name* referring to an object via dotted attribute notation prevents an exact and conflicting match.

Search Path and Path Search

The process of importing a module requires a process called a *path search*. This is the procedure of checking "predefined areas" of the file system to look for your `mymodule.py` file in order to load the `mymodule` module. These predefined areas are no more than a set of directories that are part of your Python *search path*. To avoid the confusion between the two, think of a path search as the pursuit of a file through a set of directories, the search path.

There may be times where importing a module fails:

```
>>> import xxx
Traceback (innermost last):
File "<interactive input>", line 1, in ?
ImportError: No module named xxx
```


When this error occurs, the interpreter is telling you it cannot access the requested module, and the likely reason is that the module you desire is not in the search path, leading to a path search failure. A default search path is automatically defined either in the compilation or installation process. This search path may be modified in one of two places.

One is the PYTHONPATH environment variable set in the *shell* or command-line interpreter that invokes Python. The contents of this variable consist of a colon-delimited set of directory paths. If you want the interpreter to use the contents of this variable, make sure you set or update it before you start the interpreter or run a Python script.

Once the interpreter has started, you can access the path itself which is stored in the sys module as the sys.path variable. Rather than a single string that is colon-delimited, the path has been "split" into a list of individual directory strings. Below is an example search path for a Unix machine. Your mileage will definitely vary as you go from system to system.

```
>>> sys.path
['', '/usr/local/lib/python1.5/', '/usr/local/lib/python1.5/plat-sunos5', '/usr/local/lib/python1.5/lib-tk', '/usr/local/lib/python1.5/lib-dynload']
```

Bearing in mind that this is just a list, we can definitely take our liberty with it and modify it at our leisure. If you know of a module you want to import, yet its directory is not in the search path, by all means use the list's append() method to add it to the path, like so:

```
sys.path.append('/home/wesc/py/lib')
```

Once this is accomplished, you can then load your module. As long as one of the directories in the search path contains the file, then it will be imported. Of course, this adds the directory only to the end of your search path. If you want to add it elsewhere, such as in the beginning or middle, then you have to use the insert() list method for those. In our examples above, we are updating the sys.path attribute interactively, but it will work the same way if run as a script.

Here is what it would look like if we ran into this problem interactively:

```
>>> import sys
>>> import mymodule
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError: No module named mymodule
>>>
>>> sys.path.append('/home/wesc/py/lib')
>>> sys.path
['', '/usr/local/lib/python1.5/', '/usr/local/lib/python1.5/plat-sunos5', '/usr/local/lib/python1.5/lib-tk', '/usr/local/lib/python1.5/lib-dynload', '/home/wesc/py/lib']
>>>
>>> import mymodule
>>>
```

On the flip side, you may have too many copies of a module. In the case of duplicates, the interpreter will load the first module it finds with the given name while rummaging through the search path in sequential order.

Namespaces

A *namespace* is a mapping of names (identifiers) to objects. The process of adding a name to a namespace consists of *binding* the identifier to the object (and increasing the reference count to the object by one). The Python Language Reference also includes the following definitions: "changing the mapping of a name is called *rebinding*[, and] removing a name is *unbinding*."

As briefly introduced in the last chapter, there are either two or three active namespaces at any given time during execution. These three namespaces are the local, global, and built-ins namespaces, but local namespaces come and go during execution, hence the "two or three" we just alluded to. The names accessible from these namespaces are dependent on their *loading order*, or the order in which the namespaces are brought into the system.

The Python interpreter loads the built-ins namespace first. This consists of the names in the `_builtins` module. Then the global namespace for the executing module is loaded, which then becomes the active namespace when the module begins execution. Thus we have our two active namespaces.

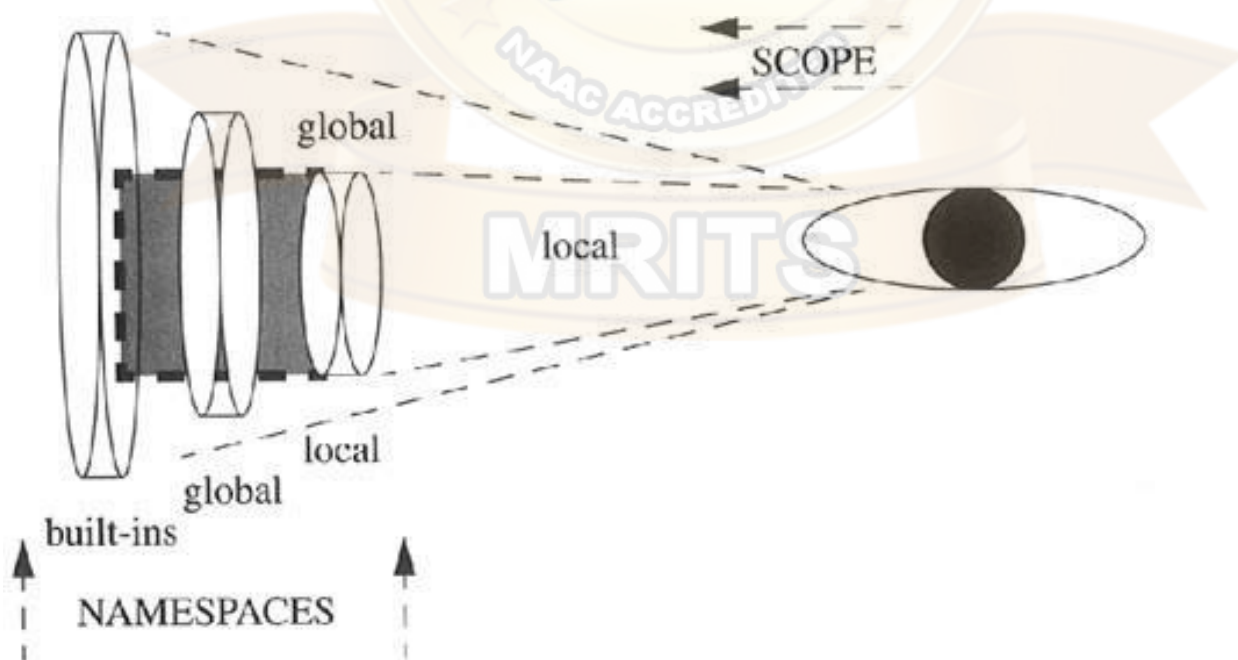
When a function call is made during execution, the third, a local, namespace is created. We can use the `globals()` and `locals()` built-in functions to tell us which names are in which namespaces. We will discuss both functions in more detail later on in this chapter.

Namespaces vs. Variable Scope

Okay, now that we know what namespaces are, how do they relate to variable scope again? They seem extremely similar. The truth is, you are quite correct.

Namespaces are purely mappings between names and objects, but scope dictates how or rather, where, one can access these names based on the physical location from within your code.

Figure 12.1. Namespaces vs. Variable Scope



Notice that each of the namespaces is a self-contained unit. But looking at the namespaces from the scoping point of view, things appear different. All names within the local namespace are within my local scope. Any name outside my local scope is in my global scope.

Also keep in mind that during the execution of the program, the local namespaces and scope are transient because function calls come and go, but the global and built-ins namespaces remain. Our final thought to you in this section is, when it comes to namespaces, ask yourself the question, "Does it have it?" And for variable scope, ask, "Can I see it?"

Name Lookup, Scoping, and Overriding

So how do scoping rules work in relationship to namespaces? It all has to do with name lookup. When accessing an attribute, the interpreter must find it in one of the three namespaces. The search begins with the local namespace. If the attribute is not found there, then the global namespace is searched. If that is also unsuccessful, the final frontier is the built-ins namespace. If the exhaustive search fails, you get the familiar:

```
>>> foo
Traceback (innermost last): File "<stdin>", line 1, in ?
NameError: foo
```

Notice how the figure features the foremost-searched namespaces "shadowing" namespaces which are searched afterwards. This is to try to convey the effect of *overriding*. This is the process whereby names may be taken out-of-scope because a more local namespace contains a name. Take a look at the following piece of code that was introduced in the previous chapter:

```
def foo():
    print "\ncalling foo()..."
    bar = 200
    print "in foo(), bar is", bar
    bar = 100
    print "in_main, bar is", bar
foo()
```

When we execute this code, we get the following output:

```
in_main, bar is 100 calling foo()...
in foo(), bar is 200
```

The bar variable in the local namespace of foo() overrode the global bar variable. Although bar exists in the global namespace, the lookup found the one in the local namespace first, hence "overriding" the global one.

21. Explain about Importing Modules.

Importing Modules

Importing a module requires the use of the **import** statement, whose syntax is:

```
import
    module1[, module2[, ... moduleN]]
```

When this statement is encountered by the interpreter, the module is imported if found in the search path. Scoping rules apply, so if imported from the top-level of a module, it has global scope; if imported from a function, it has local scope.

When a module is imported the first time, it is loaded and executed.

Module "Executed" When Loaded

One effect of loading a module is that the imported module is "executed," that is, the top-level portion of the imported module is directly executed. This usually includes setting up of global variables as well as performing the class and function declarations, and if there is a check for

name to do more on direct script invocation, that is executed too.

Of course, this type of execution may or may not be the desired effect. If not, you will have to put as much code as possible into functions. Suffice it to say that good module programming style dictates that only function and/or class definitions should be at the top-level of a module.

Importing vs. Loading

A module is loaded only once, regardless of the number of times it is imported. This prevents the module "execution" from happening over and over again if multiple imports occur. If your module imports the `sys` module, and so do five of the other modules you import, it would not be wise to load `sys` (or any other module) each time! So rest assured, loading happens only once, on first import.

Importing Module Attributes

It is possible to import specific module elements into your own module. By this, we really mean importing specific names from the module into the current namespace. For this purpose, we can use the **from-import** statement, whose syntax is:

```
from
module
import
name1[, name2[, ... nameN]]
```

Names Imported into Current Namespace

Calling **from-import** brings the name into the current namespace, meaning that you do not use the attribute/dotted notation to access the module identifier. For example, to access a variable named `var` in module `module` that was imported with:

```
from
module
import
var
```

we would use `var` by itself. There is no need to reference the module since you just imported. It is also possible to import all the names from the module into the current namespace using the following **from-import** statement:

```
from
module
import*
```


Names Imported into Importer's Scope

Another side effect of importing only names from other modules is that the names are now part of the scope of the importing module. This means that changes to the variable affect only the local copy and not the original in the imported module's namespace. In other words, the binding is now local rather than across namespaces.

Below, we present the code to two modules: an importer, `imptr.py`, and an importee, `imptee.py`. Currently, `imptr.py` uses the **from-import** statement which creates only local bindings.

```
#####
# imptee.py #
#####
foo = 'abc'
def show():
    print 'foo from imptee:', foo

#####
# imptr.py #
#####
from imptee import foo, show
show()
foo = 123
print 'foo from imptr:', foo
show()
```

Upon running the importer, we discover that the importee's view of its `foo` variable has not changed even though we modified it in the importer.

```
foo from imptee: abc
foo from imptr: 123
foo from imptee: abc
```

The only solution is to use `import` and *fully-qualified* identifier names using the attribute/dotted notation.

```
#####
# imptr.py #
#####
import imptee
imptee.show()
imptee.foo = 123
print 'foo from imptr:', imptee.foo
imptee.show()
```

Once we make the update and change our references accordingly, we now have achieved the desired effect.

```
foo from imptee: abc
foo from imptr: 123
foo from imptee: 123
```


22. List all the Module Built-in Functions and explain each.

Module Built-in Functions

The importation of modules has some functional support from the system. We will look at those now.

`_import_()`

The `_import_()` function is new as of Python 1.5, and it is the function that actually does the importing, meaning that the `import` statement invokes the `_import_()` function to do its work. The purpose of making this a function is to allow for overriding it if one is inclined to develop his or her own importation algorithm.

The syntax of `_import_()` is:

```
_import_(module_name[, globals[, locals[, fromlist]])
```

The `module_name` variable is the name of the module to import, `globals` is the dictionary of current names in the global symbol table, `locals` is the dictionary of current names in the local symbol table, and `fromlist` is a list of symbols to import the way they would be imported using the **from-import** statement.

The `globals`, `locals`, and `fromlist` arguments are optional, and if not provided, default to `globals()`, `locals()`, and `[]`, respectively.

Calling `'importsys'` can be accomplished with

```
sys = _import_('sys')
```

`globals()` and `locals()`

The `globals()` and `locals()` built-in functions return dictionaries of the global and local namespaces, respectively, of the caller. From within a function, the local namespace represents all names defined for execution of that function, which is what `locals()` will return. `globals()`, of course, will return those names globally accessible to that function.

From the global namespace, however, `globals()` and `locals()` return the same dictionary because the global namespace is as local as you can get while executing there. Here is a little snippet of code that calls both functions from both namespaces:

```
def foo():
    print '\ncalling foo()...'
    aString = 'bar'
    anInt = 42
    print "foo()'s globals:", globals().keys()
    print "foo()'s locals:", locals().keys()
    print "_main_'s globals:", globals().keys()
    print "_main_'s locals:", locals().keys()
    foo()
```

We are going to ask for the dictionary keys only because the values are of no consequence here (plus they make the lines wrap even more in this text). Executing this script, we get the following output:

```
% namespaces.py
_main_'s globals: ['_doc_', 'foo', '_name_', 'builtins']
_main_'s locals: ['_doc_', 'foo', '_name_', 'builtins']
calling foo()...
```

```
foo()'s globals: ['_doc_', 'foo', ' _name ', '_ builtins ' ]
foo()'s locals: ['anInt', 'aString']
```

reload()

The reload() built-in function performs another import on a previously imported module. The syntax of reload() is:

```
reload(module)
```

module is the actual module you want to reload. There are some criteria to using the reload() module. The first is that the module must have been imported in full (not by using **from-import**), and it must have loaded successfully. The second rule follows from the first, and that is the argument to reload() is the module itself and not a string containing the module name, i.e., it must be something like reload(sys) instead of reload('sys').

Also, code in a module is executed when it is imported, but only once. A second import does not re-execute the code, it just binds the module name. Thus reload() makes sense, as it overrides this default behavior.

23. Discuss in detail about packages.

Packages

A *package* is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages. Packages were added to Python 1.5 to aid with a variety of problems including:

- Adding hierarchical organization to flat namespace
- Allowing developers to group-related modules
- Allowing distributors to ship directories vs. bunch of files
- Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages use the standard **import** and **from-import** statements.

Directory Structure

The example package directory structure below is available .

```
Phone/
  _init_.py Voicedta/
  _init_.py Pots.py Isdn.py
  Fax/
  _init_.py G3.py
  Mobile/
  _init_.py Analog.py Digital.py
  Pager/
  _init_.py Numeric.py
```

Phone is top-level package and Voicedta, etc., are subpackages. Import subpackages by using **import** like this:

```
import Phone.Mobile.Analog Phone.Mobile.Analog.dial()
```

Alternatively, you can use **from-import** in a variety of ways:

The first way is importing just the top-level subpackage and referencing down the subpackage tree using the attribute/dotted notation:

```
from Phone import Mobile Mobile.Analog.dial('4 555-1212')
```

Further more, we can do down one more subpackage for referencing:

```
from Phone.Mobile import Analog Analog.dial('555-1212')
```

In fact, you can go all the way down in the subpackage tree structure:

```
from Phone.Mobile.Analog import dial dial('555-1212')
```

In our above directory structure hierarchy, we observe a number of `_init_.py` files. These are initializer modules that are required when using **from-import** to import subpackages, but should otherwise exist though they can remain empty.

Using from-importwith Packages

Packages also support the **from-import**all statement:

```
from package.module  
import*
```

However, such a statement is too operating system filesystem-dependent for Python to make the determination which files to import. Thus the `_all_` variable in `_init_.py` is required. This variable contains all the module names that should be imported when the above statement is invoked if there is such a thing. It consists of a list of module names as strings.

Other Features of Modules

Auto-loaded Modules

When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use. The only one that affects you is the `_builtin` module, which normally gets loaded in as the `builtins` module.

The `sys.modules` variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

For example, in Windows, the `sys.modules` variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's `keys()` method:

```
>>> import sys  
>>> sys.modules.keys()  
['os.path', 'os', 'exceptions', '_main_', 'ntpath',  
'strop', 'nt', 'sys', '_builtin_', 'site',  
'signal', 'UserDict', 'string', 'stat']
```

The loaded modules for Unix are quite similar:

```
>>> import sys  
>>> sys.modules.keys()  
['os.path', 'os', 'readline', 'exceptions',  
'_ main ___', 'posix', 'sys', 'builtin ', 'site',  
'signal', 'UserDict', 'posixpath', 'stat']
```

Preventing Attribute Import

If you do not want module attributes imported when a module is imported with "**from module import ***", begin the name, and prepend the underscore (`_`) to their names. Names in the imported module that begin with an underscore (`_`) are not imported. This minimal level of data hiding does not apply if the entire module is imported



UNIT - III

Regular Expressions: Introduction, Special Symbols and Characters, Res and Python

Multithreaded Programming: Introduction, Threads and Processes, Python, Threads, and the Global Interpreter Lock, Thread Module, Threading Module, Related Modules

1. Explain in detail about Regular Expressions.

Regular Expressions

Introduction/Motivation

Manipulating text/data is a big thing. Word processing, "fill-out-form" Web pages, streams of information coming from a database dump, stock quote information, news feeds—the list goes on and on. Because we may not know the exact text or data which we have programmed our machines to process, it becomes advantageous to be able to express this text or data in patterns which a machine can recognize and take action upon.

If I were running an electronic mail (e-mail) archiving company, and you were one of my customers who requested all his or her e-mail sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually. You would be horrified (and infuriated) that someone would be rummaging through your messages, even if his or her eyes were *supposed* to be looking only at timestamps. Regular Expressions (REs) provide such an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality. REs are simply strings which use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern ([Figure15.1](#)). In other words, they enable matching of multiple strings. Python supports REs through the standard library `re` module.

Figure 15.1. You can use regular expressions, such as the one here which recognizes valid Python identifiers. "[A-Za-z]\w+" means the first character should be alphabetic, i.e., either A–Z or a–z, followed by at least one (+) alphanumeric character (\w). In our filter, notice how many strings go into the filter, but the only ones to come out are the ones we asked for via the RE.



4xZ

"[A-Za-z]\w+" filter

Regular
Expression
Engine

OUTPUT

FOO WRITS
FOO abc 0 BS
abc MnM

First Regular Expression

As mentioned above, REs are strings containing text and special characters which describe a pattern with which to recognize multiple strings. For general text, the alphabet used for regular expressions is the set of all uppercase and lowercase letters plus numeric digits. Specialized alphabets are also possible, for instance, one consisting of only the characters "0" and "1." The set of all strings over this alphabet describes all binary strings, i.e., "0," "1," "00," "01," "10," "11," "100," etc.

Let us look at the most basic of regular expressions now to show that although REs are sometimes considered an "advanced topic," they can also be rather simplistic. Using the standard alphabet for general text, we present some simple REs and the strings which their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern which matches only one string, the string defined by the regular expression. We now present the REs followed by the strings which match them:

RE Pattern	String(s) Matched
foo	foo
Python	Python
abc123	abc123

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string which matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow a RE to match a set of strings rather than a single one.

Special Symbols and Characters for REs

We will now introduce the most popular of the *metacharacters*, special characters and symbols, which give regular expressions their power and flexibility. You will find the most common of these symbols and characters in [Table 15.1](#).

Table 15.1. Common Regular Expression Symbols and Special Characters

Notation	Description	Example RE
Symbols		
<i>re_string</i>	match literal string value <i>re_string</i>	foo
<i>re1 re2</i>	match literal string value <i>re1</i> or <i>re2</i>	foo bar
.	match any character (except NEWLINE)	::.+::
^	match start of string	^Dear
\$	match end of string	/bin/\w*sh\$
*	match 0 or more occurrences of preceding RE	[A-Za-z]\w*
+	match 1 or more occurrences of preceding RE	\d+\. \.\d+
?	match 0 or 1 occurrence(s) of preceding RE	goo?
{ <i>N</i> }	match <i>N</i> occurrences of preceding RE	\d{3}
{ <i>M, N</i> }	match from <i>M</i> to <i>N</i> occurrences of preceding RE	\d{5,9}
[...]	match any single character from character class	[aeiou]
[...x-y...]	match any single character in the range from <i>x</i> to <i>y</i>	[0-9], [A-Za-z]
[^...]	do not match any character from character class, including any ranges, if present	[^aeiou], [^A-Za-z0-9_]
(* + ? {} ?)?	apply non-greedy versions of above occurrence/repetition symbols (*, +, ?, {})	.*?\w
(...)	match enclosed RE and save as subgroup	(\d{3})?, f(oo u)bar
Special Characters		
\d	match any decimal digit, same as [0-9] (\D is inverse of \d: do not match any numeric digit)	data\d+.txt
\w	match any alphanumeric character, same as [A-Za-z0-9_] (\W is inverse of \w)	[A-Za-z_]\w+
\s	match any whitespace character, same as [\n\t\r\v\f] (\S is inverse of \s)	of\sthe
\b	match any word boundary (\B is inverse of \b)	\bThe\b
\nn	match saved subgroup nn (see (...) above)	price: \16
\c	match any special character <i>c</i> verbatim (i.e., without its special meaning, literal)	\\. \\, *
\A (\Z)	match start (end) of string (also see ^ and \$ above)	\ADear

Matching more than one RE pattern with alternation (|)

The pipe symbol (|), a vertical bar on your keyboard, indicates an *alternation* operation, meaning that it is used to choose from one of the different regular expressions which are separated by the pipe symbol. For example, below are some patterns which employ alternation, along with the strings they match:

RE Pattern	Strings Matched
at home	at, home
r2d2 c3po	r2d2, c3po
bat bet bit	bat, bet, bit

With this one symbol, we have just increased the flexibility of our regular expressions, enabling the matching of more than just one string. Alternation is also sometimes called union or logical OR.

Matching any single character (.)

The dot or period (.) symbol matches any single character except for NEWLINE (Python REs have a compilation flag [S or DOTALL] which can override this to include NEWLINES.). Whether letter, number, whitespace not including "\n," printable, non-printable, or a symbol, the dot can match them all.

RE Pattern	Strings Matched
f.o	any character between "f" and "o," e.g., fao, f9o, f#o, etc.
..	any pair of characters
.end	any character before the string end

Q: "What if I want to match the dot or period character?"

A: In order to specify a dot character explicitly, you must escape its functionality with a backslash, as in "\."

Matching from the beginning or end of strings or word boundaries (^/\$)

There are also symbols and related special characters to specify searching for patterns at the beginning and ending of strings. To match a pattern starting from the beginning, you must use the carat symbol (^) or the special character \A (backslash-capital "A"). The latter is primarily for keyboards which do not have the carat symbol, i.e., international. Similarly, the dollar sign (\$) or \Z will match a pattern from the end of a string.

Patterns which use these symbols differ from most of the others we describe in this chapter since they dictate location or position. In the Core Note above, we noted that a distinction is made between "matching," attempting matches of entire strings starting at the beginning, and "searching," attempting matches from anywhere within a string. Because we are looking specifically at symbols and special characters which deal with position, they make sense only when applied to searching.

That said, here are some examples of "edge-bound" RE search patterns:

RE Pattern	Strings Matched
^From	any string which starts with From
/bin/tcsh\$	any string which ends with /bin/tcsh
^Subject: hi\$	any string consisting solely of the string Subject: hi

Again, if you want to match either (or both) of these characters verbatim, you must use an escaping backslash. For example, if you wanted to match any string which ended with a dollar sign, one possible RE solution would be the pattern ".*\\$\$".

The `\b` and `\B` special characters will match the empty string, meaning that they can start performing the match anywhere. The difference is that `\b` will match a pattern to a word boundary, meaning that a pattern must be at the beginning of a word, whether there are any characters in front of it (word in the middle of a string) or not (word at the beginning of a line). And likewise, `\B` will match a pattern only if it appears starting in the middle of a word (i.e., not at a word boundary). Here are some examples:

RE Pattern	Strings Matched
<code>the</code>	any string containing <code>the</code>
<code>\bthe</code>	any word which starts with <code>the</code>
<code>\bthe\b</code>	matches only the word <code>the</code>
<code>\Bthe</code>	any string which contains but does not begin with <code>the</code>

Creating character classes ([])

While the dot is good for allowing matches of any symbols, there may be occasions where there are specific characters you want to match. For this reason, the bracket symbols ([]) were invented. The regular expression will match from any of the enclosed characters. Here are some examples:

RE Pattern	Strings Matched
<code>b[aeiu]t</code>	<code>bat, bet, bit, but</code>
<code>[cr][23][dp][o2]</code>	"r" or "c" then "2" or "3" followed by "d" or "p" and finally, either "o" or "2," e.g., <code>c2do, r3p2, r2d2, c3po, etc.</code>

One side note regarding the RE `"[cr][23][dp][o2]"`—a more restrictive version of this RE would be required to allow only `"r2d2"` or `"c3po"` as valid strings. Because brackets merely imply "logical OR" functionality, it is not possible to use brackets to enforce such a requirement. The only solution is to use the pipe, as in `"r2d2|c3po"`.

For single character REs, though, the pipe and brackets are equivalent. For example, let's start with the regular expression `"ab"` which matches only the string with an "a" followed by a "b." If we wanted either a one-letter string, i.e., either "a" or a "b," we could use the RE `"[ab]"`. Because "a" and "b" are individual strings, we can also choose the RE `"a|b"`. However, if we wanted to match the string with the pattern `"ab"` followed by `"cd,"` we cannot use the brackets because they work only for single characters. In this case, the only solution is `"ab|cd,"` similar to the `"r2d2/c3po"` problem just mentioned.

Denoting ranges (-) and negation (^)

In addition to single characters, the brackets also support ranges of characters. A hyphen between a pair of symbols enclosed in brackets is used to indicate a range of characters, e.g., `A-Z`, `a-z`, or `0-9` for uppercase letters, lowercase letters, and numeric digits, respectively.

This is a lexicographic range, so you are not restricted to using just alphanumeric characters. Additionally, if a caret (`^`) is the first character immediately inside the open left bracket, this symbolizes a directive to not match any of the characters in the given character set.

RE Pattern	Strings Matched
<code>z.[0-9]</code>	"z" followed by any character then followed by a single digit
<code>[r-u][env-y][us]</code>	"r" "s," "t" or "u" followed by "e," "n," "v," "w," "x," or "y" followed by "u" or "s"
<code>[^aeiou]*</code>	zero or more (*symbol introduced in next subsection) non-vowels (EXERCISE: Why do we say "non-vowels" rather than "consonants?")
<code>[^\t\n]+</code>	one or more (+symbol introduced in next subsection) characters up to, but not including, the first TAB or NEWLINE encountered
<code>["-a]</code>	in an ASCII system, all characters which fall between "" and "a," i.e., between ordinals 34 and 97.

Multiple occurrence/repetition using closure operators (*, +, ?, { })

We will now introduce the most common RE notations, namely, the special symbols *, +, and ?, all of which can be used to match single, multiple, or no occurrences of string patterns. The asterisk or star operator (*) will match zero or more occurrences of the RE immediately to its left (in language and compiler theory, this operation is known as the Kleene Closure). The plus operator (+) will match one or more occurrences of an RE (known as Positive Closure), and the question mark operator (?) will match exactly 0 or 1 occurrences of an RE.

There are also brace operators ({ }) with either a single value or a comma-separated pair of values. These indicate a match of exactly *N* occurrences (for { *N* }) or a range of occurrences, i.e., { *M*, *N* } will match from *M* to *N* occurrences. These symbols may also be escaped with the backslash, i.e., "*" matches the asterisk, etc.

Finally, the question mark (?) is overloaded so that if it follows any of the following symbols, it will direct the regular expression engine to match as few repetitions as possible.

Here are some examples using the closure operators:

RE Pattern	Strings Matched
[dn]ot?	"d" or "n," followed by an "o" and, at most, one "t" after that, i.e., do, no, dot, not
0?[1-9]	any numeric digit, possibly prepended with a "0," e.g., the set of numeric representations of the months January to September, whether single- or double-digits
[0-9]{15,16}	fifteen or sixteen digits, e.g., credit card numbers
</?[>]+>	strings which match all valid (and invalid) HTML tags
[KQRBNP][a-h][1-8]-[a-h][1-8]	Legal chess move in "long algebraic" notation (move only, no capture, check, etc.), i.e., strings which start with any of "K," "Q," "R," "B," "N," or "P" followed by a hyphenated-pair of chess board grid locations from "a1" to "h8" (and everything in between), with the first coordinate indicating the former position and the second being the new position.

Special characters representing character sets

We also mentioned that there are special characters which may represent character sets. Rather than using a range of "0-9," you may simply use "\d" to indicate the match of any decimal digit. Another special character "\w" can be used to denote the entire alphanumeric character class, serving as a shortcut for "A-Za-z0-9_," and "\s" for whitespace characters. Uppercase versions of these strings symbolizes a *non-match*, i.e., "\D" matches any non-decimal digit (same as "[^0-9]"), etc.

Using these shortcuts, we will present a few more complex examples:

RE Pattern	Strings Matched
\w+-\d+ [A-	alphanumeric string and number separated by a hyphen
Za-z]\w*	alphabetic first character, additional characters (if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers [see exercises])
\d{3}-\d{3}-\d{4}	(American) telephone numbers with an area code prefix, as in 800-555-1212
\w+@\w+\.com	simple e-mail addresses of the form XXX@YYY.com

Note that all special characters, including all the ones mentioned before such as "\A,"

"\B," "\d," etc., may or may not have ASCII equivalents. To be sure you are using the regular expression versions, it would be a safe bet to use raw strings to escape backslash functionality (see the Core Note later in this chapter). Also, the "\w" and "\W" alphanumeric character sets are affected by the L or LOCALE compilation flag and in Python 1.6 and newer, by Unicode flags.

Designating groups with parentheses ()

Now, perhaps we have achieved the goal of matching a string and discarding non-matches, but in some cases, we may also be more interested in the data that we did match. Not only do we want to know whether the entire string matched our criteria, but whether we can also extract any specific strings or substrings which were part of a successful match. The answer is yes. To accomplish this, surround any RE with a pair of parentheses.

A pair of parentheses () can accomplish either (or both) of the below when used with regular expressions:

- grouping regular expressions
- matching subgroups

One good example for wanting to group regular expressions is when you have two different REs with which you want to compare a string. Another reason is to group an RE in order to use a repetition operator on the entire RE (as opposed to an individual characters or character classes).

One side-effect of using parentheses is that the substring which matched the pattern is saved for future use. These subgroups can be recalled for the same match or search, or extracted for post-processing. Why are matches of subgroups important? The main reason is that there are times where you want to extract the patterns you match, in addition to making a match.

For example, what if we decided to match the pattern "\w+-\d+" but wanted save the alphabetic first part and the numeric second part individually? This may be desired because with any successful match, we may want to see just what those strings were that matched our RE patterns. If we add parentheses to both subpatterns, i.e., "(w+)-(\d+)," then we can access each of the matched subgroups individually. Subgrouping is preferred because the alternative is to write code to determine we have a match, then execute another separate routine (which we also had to create) to parse the entire match just to extract both parts. Why not let Python do it, since it is a supported feature of the re module, instead of "reinventing the wheel"?

RE Pattern	Strings Matched
\d+(\.\d*)?	strings representing simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc.
(Mr?s?\ .)?[A-Z][a-z]* [A-Za-z-]+	first name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters

REs and Python

The re module was introduced to Python in version 1.5. If you are using an older version of Python, you will have to use the now-obsolete regex and reesub modules—these older modules are more Emacs-flavored, are not as full-featured, and are in many ways incompatible with the current re module.

However, regular expressions are still regular expressions, so most of the basic concepts from this section can be used with the old regex and reesub software. In contrast, the new re module supports the more powerful and regular Perl-style (Perl5) REs, allows multiple threads to share the same compiled RE objects, and supports named subgroups. In addition, there is a transition module called reconvert to help developers move from regex/reesub to re.

The re engine was rewritten in 1.6 for performance enhancements as well as adding Unicode support. The interface was not changed, hence the reason the module name was left alone. The new re engine—known internally as sre—thus replaces the existing 1.5 engine—internally called pcre.

reModule: Core Functions and Methods

The chart in [Table 15.2](#) lists the more popular functions and methods from the re module. Many of these functions are also available as methods of compiled regular expression objects "regex objects" and RE "match objects." In this subsection, we will look at the two main functions/methods, match() and search(), as well as the compile() function. We will introduce several more in the next section, but for more information on all these and the others which we do not cover, we refer you to the Python documentation.

<i>Function/Method</i>	<i>Description</i>
re Module Function Only	
<code>compile(pattern, flags=0)</code>	compile RE <i>pattern</i> with any optional <i>flags</i> and return a regex object
re Module Functions and regex Object Methods	
<code>match(pattern, string, flags=0)</code>	attempt to match RE <i>pattern</i> to <i>string</i> with optional <i>flags</i> ; return match object on success, None on failure
<code>search(pattern, string, flags=0)</code>	search for first occurrence of RE <i>pattern</i> within <i>string</i> with optional <i>flags</i> ; return match object on success, None on failure
<code>findall(pattern, string)</code>	look for all (non-overlapping) occurrences of <i>pattern</i> in <i>string</i> ; return a list of matches (new as of Python 1.5.2)

Table 15.2. Common Regular Expression Functions and Methods

<code>split(pattern, string, max=0)</code>	split <i>string</i> into a list according to RE <i>pattern</i> delimiter and return list of successful matches, splitting at most <i>max</i> times (split all occurrences is the default)
<code>sub(pattern, repl, string, max=0)</code>	replace all occurrences of the RE <i>pattern</i> in <i>string</i> with <i>repl</i> , substituting all occurrences unless <i>max</i> provided (also see <code>subn()</code> which, in addition, returns the number of substitutions made)
Match Object Methods	
<code>group(num=0)</code>	return entire match (or specific subgroup <i>num</i>)
<code>groups()</code>	return all matching subgroups in a tuple (empty if there weren't any)

Compiling REs with compile()

Optional flags may be given as arguments for specialized compilation. These flags allow for case-insensitive matching, using system locale settings for matching alphanumeric characters, etc. Please refer to the documentation for more details. These flags, some of which have been briefly mentioned (i.e. DOTALL, LOCALE), may also be given to the module versions of match() and search() for a specific pattern match attempt—these flags are mostly for compilation reasons, hence the reason why they can be passed to the module versions of match() and search() which do compile an RE pattern once. If you want to use these flags with the methods, they must already be integrated into the compiled regex objects.

In addition to the methods below, regex objects also have some data attributes, two of which include any compilation flags given as well as the regular expression pattern compiled.

Match objects and the group() and groups() Methods

There is another object type in addition to the regex object when dealing with regular expressions, the *match object*. These objects are those which are returned on successful calls to match() or search(). Match objects have two primary methods, group() and groups().

group() will either return the entire match, or a specific subgroup, if requested. groups() will simply return a tuple consisting of only/all the subgroups. If there are no subgroups requested, then groups() returns an empty tuple while group() still returns the entire match.

Matching strings with match()

match() is the first re module function and RE object (regex object) method we will look at. The match() function attempts to match the pattern to the string, starting at the beginning. If the match is successful, a match object is returned, but on failure, None is returned. The group() method of a match object can be used to show the successful match. Here is an example of how to use match()[and group()]:

```
>>> m = re.match('foo', 'foo')    # pattern matches string
>>> if m != None:                  # show match if successful
...     m.group()
...
'foo'
```

The pattern "foo" matches exactly the string "foo." We can also confirm that m is an example of a match object from within the interactive interpreter:

```
>>> m    # confirm match object returned
<re.MatchObject instance at 80ebf48>
```

Here is an example of a failed match where None is returned:

```
>>> m = re.match('foo', 'bar')    # pattern does not match string
>>> if m != None: m.group()        # (1-line version of if clause)
...
>>>
```

The match above fails, thus None is assigned to m, and no action is taken due to the way we constructed our if statement. For the remaining examples, we will try to leave out the if check for brevity, if possible, but in practice it is a good idea to have it there to prevent AttributeError exceptions.

A match will still succeed even if the string is longer than the pattern as long as the pattern matches from the beginning of the string. For example, the pattern "foo" will find a match in the string "food on the table" because it matches the

pattern from the beginning:

```
>>> m = re.match('foo', 'food on the table') # match succeeds
>>> m.group()
'foo'
```

As you can see, although the string is longer than the pattern, a successful match was made from the beginning of the string. The substring "foo" represents the match which was extracted from the larger string.

We can even sometimes bypass saving the result altogether, taking advantage of Python's object-oriented nature:

```
>>> re.match('foo', 'food on the table').group()
'foo'
```

Note from a few paragraphs above that an `AttributeError` will be generated on a non-match.

Looking for a pattern within a string with `search()`(searching vs. matching)

The chances are greater that the pattern you seek is somewhere in the middle of a string, rather than at the beginning. This is where `search()` comes in handy. It works exactly in the same way as `match` except that it searches for the first occurrence of the given RE pattern anywhere with its string argument. Again, a match object is returned on success and `None` otherwise.

We will now illustrate the difference between `match()` and `search()`. Let us try a longer string match attempt. This time, we will try to match our string "foo" to "seafood":

```
>>> m = re.match('foo', 'seafood') # no match
>>> if m != None: m.group()
...
>>>
```

As you can see, there is no match here. `match()` attempts to match the pattern to the string from the beginning, i.e., the "f" in the pattern is matched against the "s" in the string, which fails immediately. However, the string "foo" *does* appear (elsewhere) in "seafood," so how do we get Python to say "yes?" The answer is by using the `search()` function. Rather than attempting a *match*, `search()` looks for the first occurrence of the pattern within the string. `search()` searches strictly from left to right.

```
>>> m = re.search('foo', 'seafood') # use search() instead
>>> if m != None: m.group()
...
'foo' # search succeeds where match failed
>>>
```

We will be using the `match()` and `search()` regex object methods and the `group()` and `groups()` match object methods for the remainder of this subsection, exhibiting a broad range of examples of how to use regular expressions with Python. We will be using almost all of the special characters and symbols which are part of the regular expression syntax.

Matching more than one string (|)

Here is how we would use that RE with Python:

```
>>> bt = 'bat|bet|bit' # RE pattern: bat, bet, bit
>>> m = re.match(bt, 'bat') # 'bat' is a match
>>> if m != None: m.group()
... 'bat'
>>> m = re.match(bt, 'blt') # no match for 'blt'
>>> if m != None: m.group()
...
>>> m = re.match(bt, 'He bit me!') # does not match string
```



```

>>> if m != None: m.group()
...
>>> m = re.search(bt, 'He bit me!')# found \qbit\q via search
>>> if m != None: m.group()
... 'bit'

```

Matching any single character (.)

In the examples below, we show that a dot cannot match a NEWLINE or a non-character, i.e., the empty string:

```

>>> anyend = '.end'
>>> m = re.match(anyend, 'bend')    # dot matches 'b'
>>> if m != None: m.group()
... 'bend'
>>> m = re.match(anyend, 'end')     # no char to match
>>> if m != None: m.group()
...
>>> m = re.match(anyend, '\nend')   # any char except \n
>>> if m != None: m.group()
...
>>> m = re.search('.end', 'The end.') # matches ' ' in search
>>> if m != None: m.group()
...
'end'

```

Below is an example of searching for a real dot (decimal point) in a regular expression where we escape its functionality with a backslash:

```

>>> patt314 = '3.14'                # RE dot
>>> pi_patt = '3\.14'                # literal dot (dec. point)
>>> m = re.match(pi_patt, '3.14')    # exact match
>>> if m != None: m.group()
... '3.14'
>>> m = re.match(patt314, '3014')    # dot matches '0'
>>> if m != None: m.group()
... '3014'
>>> m = re.match(patt314, '3.14')    # dot matches '.'
>>> if m != None: m.group()
... '3.14'

```

Creating character classes ([])

Earlier, we had a long discussion regarding "[cr][23][dp][o2]" and how it differs from "r2d2|c3po". With the examples below, we will show that "r2d2|c3po" is more restrictive than "[cr][23][dp][o2]":

```

>>> m = re.match('[cr][23][dp][o2]', 'c3po')# matches \qc3po\q
>>> if m != None: m.group()
... 'c3po'
>>> m = re.match('[cr][23][dp][o2]', 'c2do')# matches 'c2do'
>>> if m != None: m.group()
... 'c2do'
>>> m = re.match('r2d2|c3po', 'c2do')# does not match 'c2do'

```



```

>>> if m != None: m.group()
...
>>> m = re.match('r2d2|c3po', 'r2d2')# matches 'r2d2'
>>> if m != None: m.group()
... 'r2d2'

```

Repetition, special characters, and grouping

The most common aspects of REs involve the use of special characters, multiple occurrences of RE patterns, and using parentheses to group and extract submatch patterns. One particular RE we looked at related to simple e-mail addresses ("`\w+@\w+\.com`"). Perhaps we want to match more e-mail addresses than this RE allows. In order to support an additional hostname in front of the domain, i.e., "`www.xxx.com`" as opposed to accepting only "`xxx.com`" as the entire domain, we have to modify our existing RE. To indicate that the hostname is optional, we create a pattern which matches the hostname (followed by a dot), use the `?` operator indicating zero or one copy of this pattern, and insert the optional RE into our previous RE as follows: "`\w+@(\w+\.)?\w+\.com`". As you can see from the examples below, either one or two names are now accepted in front of the ".com".

```

>>> patt = '\w+@(\w+\.)?\w+\.com'
>>> re.match(patt, 'nobody@xxx.com').group() 'nobody@xxx.com'
>>> re.match(patt, 'nobody@www.xxx.com').group() 'nobody@www.xxx.com'

```

Furthermore, we can even extend our example to allow any number of intermediate subdomain names with the following pattern: "`\w+@(\w+\.)*\w+\.com`":

```

>>> patt = '\w+@(\w+\.)*\w+\.com'
>>> re.match(patt, 'nobody@www.xxx.yyy.zzz.com').group() 'nobody@www.xxx.yyy.zzz.com'

```

However, we must add the disclaimer that using solely alphanumeric characters does not match all the possible characters which may make up e-mail addresses. The above RE patterns would not match a domain such as "`xxx-yyy.com`" or other domains with "`\W`" characters.

Earlier, we discussed the merits of using parentheses to match and save subgroups for further processing rather than coding a separate routine to manually parse a string after an RE match had been determined. In particular, we discussed a simple RE pattern of an alphanumeric string and a number separated by a hyphen, "`\w+-\d+`," and how adding subgrouping to form a new RE, "`(\w+)-(\d+)`," would do the job. Here is how the original RE works:

```

>> m = re.match('\w\w\w-\d\d\d', 'abc-123')
>>> if m != None: m.group()
...
'abc-123'
>>> m = re.match('\w\w\w-\d\d\d', 'abc-xyz')
>>> if m != None: m.group()
...
>>>

```

In the above code, we created an RE to recognize three alphanumeric characters followed by three digits. Testing this RE on "`abc-123`," we obtained with positive results while

"`abc-xyz`" fails. We will now modify our RE as discussed before to be able to extract the alphanumeric string and number. Note how we can now use the `group()` method to access individual subgroups or the `groups()` method to obtain a tuple of all the subgroups matched:

```

>>> m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')
>>> m.group()           # entire match 'abc-123'
>>> m.group(1)         # subgroup 1 'abc'
>>> m.group(2)         # subgroup 2 '123'

```

```
>>> m.groups() # all subgroups ('abc', '123')
```

As you can see, group() is used in the normal way to show the entire match, but can also be used to grab individual subgroup matches. We can also use the groups() method to obtain a tuple of all the substring matches.

Here is a simpler example showing different group permutations, which will hopefully make things even more clear:

```
>>> m = re.match('ab', 'ab') # no subgroups
>>> m.group() # entire match
'ab'
>>> m.groups() # all subgroups
()
>>>
>>> m = re.match('(ab)', 'ab') # one subgroup
>>> m.group() # entire match
'ab'
>>> m.group(1) # subgroup 1
'ab'
>>> m.groups() # all subgroups
('ab',)
>>>
>>> m = re.match('(a)(b)', 'ab') # two subgroups
>>> m.group() # entire match
'ab'
>>> m.group(1) # subgroup 1
'a'
>>> m.group(2) # subgroup 2 'b'
'b'
>>> m.groups() # all subgroups ('a', 'b')
('a', 'b')
>>>
>>> m = re.match('(a(b))', 'ab') # two subgroups
>>> m.group() # entire match 'ab'
'ab'
>>> m.group(1) # subgroup 1 'ab'
'ab'
>>> m.group(2) # subgroup 2 'b'
'b'
>>> m.groups() # all subgroups ('ab', 'b')
('ab', 'b')
```

Matching from the beginning and end of strings and on word boundaries

The following examples highlight the positional RE operators. These apply more for searching than matching because match() always starts at the beginning of a string.

```
>>> m = re.search('^The', 'The end.') # match
>>> if m != None: m.group()
... 'The'
>>> m = re.search('^The', 'end. The') # not at beginning
>>> if m != None: m.group()
...
>>> m = re.search(r'\bthe', 'bite the dog') # at a boundary
>>> if m != None: m.group()
... 'the'
```

```

>>> m = re.search(r'\bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
...
>>> m = re.search(r'\Bthe', 'bitethe dog') # no boundary
>>> if m != None: m.group()
... 'the'

```

You will notice the appearance of raw strings here. You may want to take a look at the Core Note towards the end of the chapter for clarification on why they are here. In general, it is a good idea to use raw strings with regular expressions.

Other reModule Functions and Methods

There are four other re module functions and regex object methods which we think you should be aware of: findall(), sub(), subn(), and split().

Finding every occurrence with findall()

findall() is new to Python as of version 1.5.2. It looks for all non-overlapping occurrences of an RE pattern in a string. It is similar to search() in that it performs a string search, but it differs from match() and search() in that findall() always returns a list. The list will be empty if no occurrences are found but if successful, it will consist of all matches found (grouped in left-to-right order of occurrence).

```

>>> re.findall('car', 'car')['car']
>>> re.findall('car', 'scary')['car']
>>> re.findall('car', 'carry the barcardi to the car')['car', 'car', 'car']

```

Subgroup searches result in a more complex list returned, and that makes sense, because subgroups are a mechanism which will allow you to extract specific patterns from within your single regular expression, such as matching an area code which is part of a complete telephone number, or a login name which is part of an entire e-mail address.

For a single successful match, each subgroup match is a single element of the resulting list returned by findall(); for multiple successful matches, each subgroup match is a single element in a tuple, and such tuples (one for each successful match) are the elements of the resulting list. This part may sound confusing at first, but if you try different examples, it will help clarify things.

Searching and replacing with sub()[and subn()]

There are two functions/methods for search-and-replace functionality: sub() and subn(). They are both almost identical and replace all matched occurrences of the RE pattern in a string with some sort of replacement. The replacement is usually a string, but it can also be a function which returns a replacement string. subn() is exactly the same as sub(), but it also returns the total number of substitutions made—both the newly-substituted string and the substitution count are returned as a 2-tuple.

```

>>> re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n') 'attn: Mr. Smith\n\nDear Mr. Smith,\n'
>>>
>>> re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n') ('attn: Mr. Smith\n\nDear Mr. Smith,\n', 2)
>>>
>>> print re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')
attn: Mr. Smith

```

Dear Mr. Smith,

```

>>> re.sub('[ae]', 'X', 'abcdef') 'XbcdXf'
>>> re.subn('[ae]', 'X', 'abcdef')

```

('XbcdXf', 2)

Splitting (on delimiting pattern) with split()

The re module and RE object method split() work similar to its string counterpart, but rather than splitting on a fixed string, it splits a string based on an RE pattern, adding some significant power to string splitting capabilities. If you do not want the string split for every occurrence of the pattern, you can specify the maximum number of splits by setting a value (other than zero) to the maxargument.

If the delimiter given is not a regular expression which uses special symbols to match multiple patterns, then re.split() works in exactly the same manner as string.split(), as illustrated in the example below (which splits on a single colon):

```
>>> re.split(':', 'str1:str2:str3')
['str1', 'str2', 'str3']
```

But with regular expressions involved, we have an even more powerful tool. Take, for example, the output from the Unix who command, which lists all the users logged into a system:

```
% who
wesc      console      Jun   20  20:33
wesc      pts/9        Jun   22  01:38      (192.168.0.6)
wesc      pts/1        Jun   20  20:33      (:0.0)
wesc      pts/2        Jun   20  20:33      (:0.0)
wesc      pts/4        Jun   20  20:33      (:0.0)
wesc      pts/3        Jun   20  20:33      (:0.0)
wesc      pts/5        Jun   20  20:33      (:0.0)
wesc      pts/6        Jun   20  20:33      (:0.0)
wesc      pts/7        Jun   20  20:33      (:0.0)
wesc      pts/8        Jun   20  20:33      (:0.0)
```

Perhaps we want to save some user login information such as login name, teletype they logged in at, when they logged in, and from where. Using string.split() on the above would not be effective, since the spacing is erratic and inconsistent. The other problem is that there is a space between the month, day, and time for the login timestamps. We would probably want to keep these fields together.

You need some way to describe a pattern such as, "split on two or more spaces." This is easily done with regular expressions. In no time, we whip up the RE pattern "\s\s+," which does mean at least two whitespace characters. Let's create a program called rewho.py that reads the output of the who command, presumably saved into a file called whodata.txt. Our rewho.pyscript initially looks something like this:

```
import re
f = open('whodata.txt', 'r')
for eachLine in f.readlines():
    print re.split('\s\s+', eachLine)
f.close()
```

We will now execute the who command, saving the output into whodata.txt, and then call rewho.py and take a look at the results:

```
% who > whodata.txt
% rewho.py
['wesc', 'console', 'Jun 20 20:33\012']
['wesc', 'pts/9', 'Jun 22 01:38\011(192.168.0.6)\012']
['wesc', 'pts/1', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/2', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/4', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/3', 'Jun 20 20:33\011(:0.0)\012']
['wesc', 'pts/5', 'Jun 20 20:33\011(:0.0)\012']
```



```
['wesc', 'pts/6', 'Jun 20 20:33\011(:0.0)\012']
```

```
['wesc', 'pts/7', 'Jun 20 20:33\011(:0.0)\012']
```

```
['wesc', 'pts/8', 'Jun 20 20:33\011(:0.0)\012']
```

It was a good first try, but not quite correct. For one thing, we did not anticipate a single TAB (ASCII \011) as part of the output (which looked like at least 2 spaces, right?), and perhaps we aren't really keen on saving the NEWLINE (ASCII \012) which terminates each line. We are now going to fix those problems as well as improve the overall quality of our application by making a few more changes.

First, we would rather run the who command from within the script, instead of doing it externally and saving the output to a whodata.txt file—doing this repeatedly gets tiring rather quickly. To accomplish invoking another program from within ours, we call upon the os.popen() command, discussed briefly in [Section 14.5.2](#). Although os.popen() is available only on Unix systems, the point is to illustrate the functionality of re.split(), which is available on all platforms.

We shall also employ the map() built-in function along with string.strip() to get rid of the trailing NEWLINES. Finally, we will add the detection of a single TAB as an additional, alternative re.split() delimiter by adding it to the regular expression. Presented below in [Example 15.1](#), is the final version of our rewho.pyscript:

Example 15.1. Split Output of Unix whoCommand (rewho.py)

This script calls the who command and parses the input by splitting up its data along various types of whitespace characters.

```
<$npage>
001 1      #!/usr/bin/env python
002 2
03   3      from os import popen
04   4      from re import split
05   5      from string import strip
06   6
07   7      f = popen('who', 'r')
08   8      for eachLine in map(strip, f.readlines()):
09   9          print split('\s+\t', eachLine)
10  10 f.close()
11  <$npage>
```

Running this script, we now get the following (correct) output:

```
% rewho.py
['wesc', 'console', 'Jun 20 20:33']
['wesc', 'pts/9', 'Jun 22 01:38', '(192.168.0.6)']
['wesc', 'pts/1', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/2', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/4', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/3', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/5', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/6', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/7', 'Jun 20 20:33', '(:0.0)']
['wesc', 'pts/8', 'Jun 20 20:33', '(:0.0)']
```

A similar exercise can be achieved in a DOS/Windows environment using the dir command in place of who.

Regular Expression Adventures

We will now run through an in-depth example of the different ways of using regular expressions for string manipulation. The first step is to come up with some code that actually generates some random (but-not-so-random) data on which to operate. In [Example 15.2](#), we present `gendata.py`, a script which generates a data set. Although this program simply displays the generated set of strings to standard output, this output may very well be redirected to a test file.

NOTE

Unix systems, as well as others, use architecture-size integers to represent the current time in seconds. Since most systems today are 32-bit, the total amount of time recognized by any platform using this mechanism is 2^{32} seconds. Such integers are signed, so we really only have $2^{31}-1$ seconds.

The current time is recognized as the number of seconds which have elapsed since time zero, which is pegged at midnight, January 1, 1970. Moving forward to the maximum possible positive 32-bit signed integer ($2^{31} - 1$), we arrive at the "end of time," which evaluates to Tuesday morning, January 19, 2038 at 3:14 AM and 7 seconds using Universal Coordinated Time (UTC/GMT). Hopefully by then, we would have discontinued the use of 32-bit systems. This phenomena is otherwise known as the Y2038 problem.)

Here is one way you could find out what the special date/time it is for your local time, using Python:

```
>>> import sys, time
>>> time.asctime(time.localtime(sys.maxint))    # Pacific Time 'Mon Jan 18 19:14:07 2038'
```

`sys.maxint` has the last possible second using a 32-bit integer. We feed that time in seconds to `time.localtime()` to obtain the tuple for your/our local time (here we are on Pacific Time), and finally, we ship that tuple off to `time.asctime()` to obtain the standard timestamp for the last possible second. As you can see from our example, we are eight hours west of the Prime/Greenwich Meridian.

This is not as much a Python Core Note as it is a general programming note, but should be nevertheless discussed for common knowledge since it applies to all 32-bit systems with applications using on the C language, regardless of platform, i.e., UNIX and non-UNIX, which use UNIX-style dating. In the `gendata.py` script coming up, we randomly generate integers, effectively generating random dates for our application.

This script generates strings with three fields, delimited by a pair of colons, or a double-colon. The first field is a random (32-bit) integer, which is converted to a date (see the accompanying Core Note). The next field is a randomly-generated electronic mail (e-mail) address, and the final field is a set of integers separated by a single dash (-).

Example 15.2. Data Generator for RE Exercises (`gendata.py`)

Create random data for regular expressions practice and output the generated data to the screen.

```
<$nopcode>
001 1      #!/usr/bin/env python
002 2
003 3      from random import randint,choice
004 4      from string import lowercase
005 5      from sys import maxint
006 6      from time import ctime
007 7
008 8      doms = ('com', 'edu', 'net', 'org', 'gov')009 9
```

```

10 10  for i in range(randint(5, 10)):
11 11      dtint = randint(0, maxint-1)                # pick date
12 012 12  dtstr = ctime(dtint)                       # date string
13 013 13
14 14      shorter = randint(4, 7)                    # login shorter
15 15      em = "
16 16      for j in range(shorter):                  # generate login
17 17          em = em + choice(lowercase)
18 18
19 19      longer = randint(shorter, 12)              # domain longer
20 20      dn = "
21 21      for j in range(longer):                  # create domain
22 22          dn = dn + choice(lowercase)
23 23
24 24  print '%s::%s@%s.%s::%d-%d-%d' % (dtstr, em,
25 25      dn, choice(doms), dtint, shorter, longer)
26  <$nopage>

```

Running this code, we get the following output (your mileage will definitely vary) and store locally as the file redata.txt:

```

Thu Jul 22 19:21:19 2004::izsp@dicqdhytvhv.edu::1090549279-4-11
Sun Jul 13 22:42:11 2008::zqeu@dxaijgkniy.com::1216014131-4-11
Sat May 5 16:36:23 1990::fclihw@alwdbzpsdg.edu::641950583-6-10
Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
Thu Jun 26 19:08:59 2036::ugxfugt@jkhuqhs.net::2098145339-7-7
Tue Apr 10 01:04:45 2012::zkwaq@rpxwmtikse.com::1334045085-5-10

```

You may or may not be able to tell, but the output from this program is ripe for regular expression processing. Following our line-by-line explanation, we will implement several REs to operate on this data, as well as leave plenty for the end-of-chapter exercises.

Line-by-line explanation

Lines 1 – 6

In our example script, we require the use of multiple modules. But since we are utilizing only one or two functions from these modules, rather than importing the entire module, we choose in this case to import only specific attributes from these modules. Our decision to use **from-import** rather than **import** was based solely on this reasoning. The **from-import** lines succeed the UNIX start-up directive on line 1.

Line 8

doms is simply a set of higher-level domain names from which we will randomly pick for each randomly-generated e-mail address.

Lines 10–12

Each time gendata.py executes, between 5 and 10 lines of output are generated. (Our script uses the random.randint() function for all cases where we desire a random integer.) For each line, we choose a random integer from the entire possible range (0 to $2^{31} - 1$ [sys.maxint]), then convert that integer to a date using time.ctime().

Lines 14–22

The login name for the fake e-mail address should be between 4 and 7 characters in length. To put it together, we randomly choose between 4 and 7 random lowercase letters, concatenating each letter to our string one-at-a-time. The functionality of the `random.choice()` function is given a sequence, return a random element of that sequence. In our case, the sequence is the set of all 26 lowercase letters of the alphabet, `string.lowercase`.

We decided that the main domain name for the fake e-mail address should be between 4 and 12 characters in length, but at least as long as the login name. Again, use random lowercase letters to put this name together letter-by-letter.

Line 24–25

The key component of our script puts together all of the random data into the output line. The date string comes first, followed by the delimiter. We then put together the random e-mail address by concatenating the login name, the "@" symbol, the domain name, and a randomly chosen high-level domain. After the final double-colon, we put together a random integer string using the original time chosen (for the date string), followed by the lengths of the login and domain names, all separated by a single hyphen.

Matching a string

For the following exercises, create both permissive and restrictive versions of your REs. We recommend you test these REs in a short application which utilizes our sample `redata.txt` file above (or use your own generated data from running `gendata.py`). You will need to use it again when you do the exercises.

To test the RE before putting it into our little application, we will import the `re` module and assign one sample line from `redata.txt` to a string variable `data`. These statements are constant across both illustrated examples.

```
>>> import re
>>> data = Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::1171590364-6-8
```

In our first example, we will create a regular expression to extract (only) the days of the week from the timestamps from each line of the data file `redata.txt`. We will use the following RE:

```
"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"
```

This example requires that the string start with ("^" RE operator) any of the seven strings listed. If we were to "translate" the above RE to English, it would read something like, "the string should start with "Mon," "Tue,"... , "Sat," or "Sun." Alternatively, we can bypass all the carat operators with a single carat if we group the day strings like this:

```
"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"
```

The parentheses around the set of strings mean that one of these strings must be encountered for a match to succeed. This is a "friendlier" version of the original RE which we came up with which did not have the parentheses. Using our modified RE, we can take advantage of the fact that we can access the matched string as a subgroup:

```
>>> patt = '^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)'
>>> m = re.match(patt, data)
>>> m.group()           # entire match 'Thu'
>>> m.group(1)         # subgroup 1 'Thu'
>>> m.groups()         # all subgroups ('Thu',)
```

This feature may not seem as revolutionary as we have made it out to be for this example, but it definitely advantageous in the next example or anywhere you provide extra data as part of the RE to help in the string matching process, even though those characters may not be part of the string you are interested in.

Both of the above REs are the most restrictive, specifically requiring a set number of strings. This may not work well in an internationalization environment where localized days and abbreviations are used. A looser RE would be: `^\w{3}`. This one requires only that a string begin with three consecutive alphanumeric characters. Again, to translate the RE into English, the carat indicates "begins with," the `\w` means any single alphanumeric character, and the `{3}` means that there should be 3 consecutive copies of the RE which the `{3}` embellishes. Again, if you want grouping, parentheses should be used, i.e., `^(w{3})`:

```
>>> patt = '^(w{3})'
>>> m = re.match(patt, data)
>>> if m != None: m.group()
... 'Thu'
>>> m.group(1) 'Thu'
```

Note that an RE of `^(w){3}` is not correct. When the `{3}` was inside the parentheses, the match for 3 consecutive alphanumeric characters was made first, then represented as a group. But by moving the `{3}` outside, it is now equivalent to 3 consecutive single alphanumeric characters:

```
>>> patt = '^(w){3}'
>>> m = re.match(patt, data)
>>> if m != None: m.group()
... 'Thu'
>>> m.group(1) 'u'
```

The reason why only the "u" shows up when accessing subgroup 1 is that subgroup 1 was being continually replaced by the next character. In other words, `m.group(1)` started out as "T," then changed to "h," then finally was replaced by "u." These are 3 individual (and overlapping) groups of a single alphanumeric character, as opposed to a single group consisting of 3 consecutive alphanumeric characters.

In our next (and final) example, we will create a regular expression to extract the numeric fields found at the end of each line of `redata.txt`.

Search vs. Match

Before we create any REs, however, we realize that these integer data items are at the end of the data strings. This means that we have a choice of using either search or match. Initiating a search makes more sense because we know exactly what we are looking for (set of 3 integers), that what we seek is not at the beginning of the string, and that it does not make up the entire string. If we were to perform a match, we would have to create an RE to match the entire line and use subgroups to save the data we are interested in. To illustrate the differences, we will perform a search first, then do a match to show you that searching is more appropriate.

Since we are looking for 3 integers delimited by hyphens, we create our RE to indicate as such:

`\d+-\d+-\d+`. This regular expression means, "any number of digits (at least one, though) followed by a hyphen, then more digits, another hyphen, and finally, a final set of digits. We test our RE now using `search()`:

```
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group() # entire match '1171590364-6-8'
```

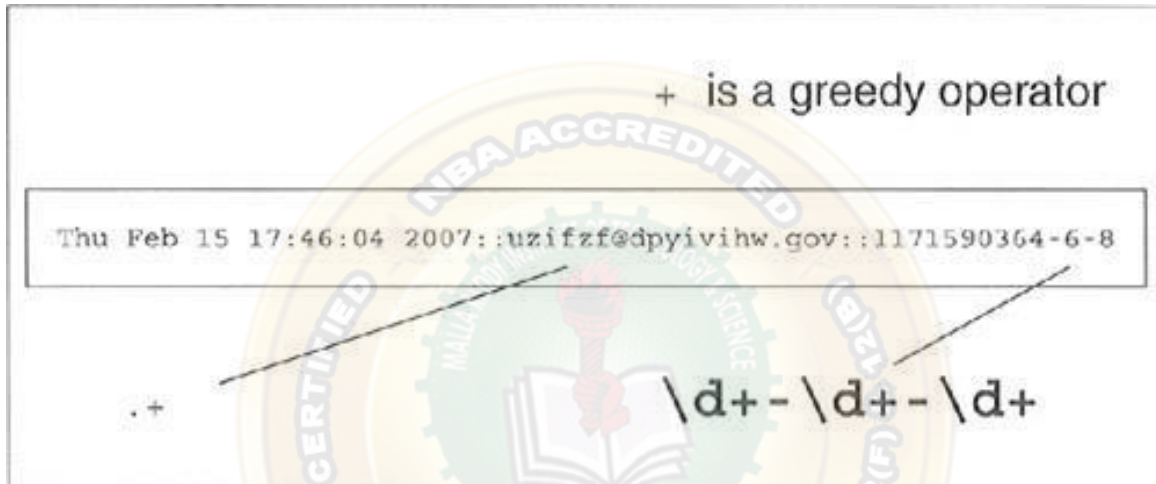
A match attempt, however, would fail. Why? Because matches start at the beginning of the string, the numeric strings are at the rear. We would have to create another RE to match the entire string. We can be lazy though, by using `.*` to indicate just an arbitrary set of characters followed by what we are really interested in:

```
patt = '.*\d+-\d+-\d+'
>>> re.match(patt, data).group() # entire match 'Thu Feb 15 17:46:04
2007::uzifzf@dpyivihw.gov::1171590364-6-8'
```

This works great, but we really want the number fields at the end, not the entire string, so we have to use parentheses to group what we want:

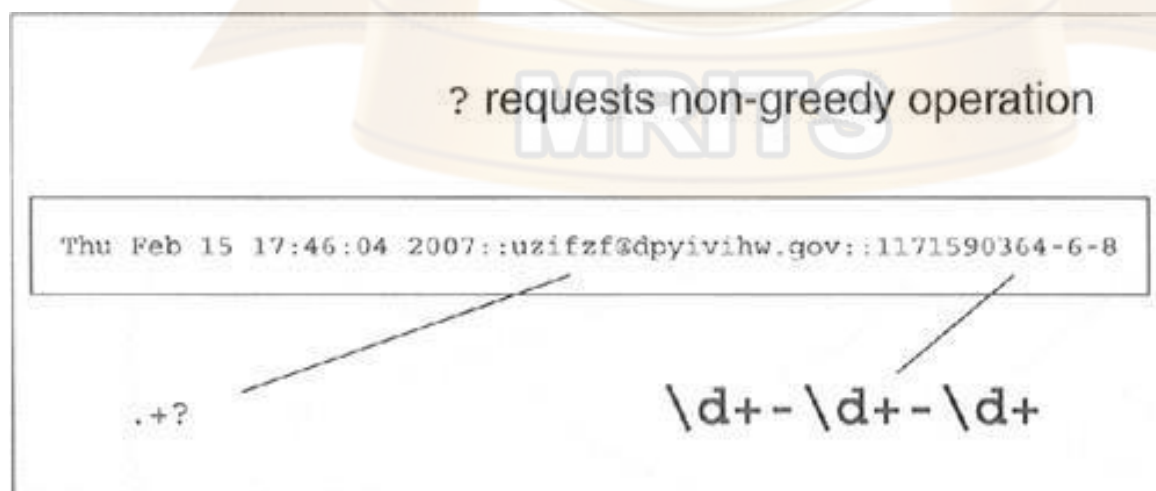

```
>>> patt = '+(\\d+-\\d+-\\d+)'
>>> re.match(patt, data).group(1)           # subgroup 1 '4-6-8'
```

We should have extracted "1171590364-6-8," not just "4-6-8". Where is the rest of the first integer? The problem is that regular expressions are inherently "greedy." That means that with wildcard patterns, regular expressions are evaluated in left-to-right order and try to "grab" as many characters as possible which match the pattern. In our case above, the "+" grabbed every single character from the beginning of the string, including most of the first integer field we wanted. The "\\d+" needed only a single digit, so it got "4," while the "+" matched everything from the beginning of the string up to that first digit: "Thu Feb 15 17:46:04 2007::uzifzf@dpyivihw.gov::117159036," as indicated below in [Figure 15-2](#).



The solution is to use the "don't be greedy" operator, "?". It can be used after "*", "+", or "?". This directs the regular expression engine to match as few characters as possible. So if we place a "?" after the "+", we obtain the desired result illustrated in [Figure 15-3](#).

Figure 15.3. Solving the Greedy Problem: Requests Non-Greediness



```
>>> patt = '+(\\d+-\\d+-\\d+)?'
>>> re.match(patt, data).group(1)           # subgroup 1 '1171590364-6-8'
```

One final example. Let's say we want to pull out only the middle integer of the three-integer field. Here is how we would do it (using a search so we don't have to match the entire string): "`-(\d+)-`". Trying out this pattern, we get:

```
>>> patt = '-(\d+)-'  
>>> m = re.search(patt, data)  
>>> m.group()           # entire match '-6-'  
>>> m.group(1)         # subgroup 1 '6'
```

We barely touched upon the power of regular expressions, and in this limited space we have not been able to give them justice. However, we hope that we have given an informative introduction so that you can add this powerful tool to your programming skills. We suggest you refer to the documentation for more details on how to use REs with Python. For more complete immersion into the world of regular expressions.

2. What is Multithreading? Explain about Multithreaded programming with suitable example.

Multithreaded Programming

Introduction/Motivation

Before the advent of multithreaded (MT) programming, running of computer programs consisted of a single sequence of steps which were executed in synchronous order by the host's central processing unit (CPU). This style of execution was the norm whether the task itself required the sequential ordering of steps or if the entire program was actually an aggregation of multiple subtasks. What if these subtasks were independent, having no causal relationship (meaning that results of subtasks do not affect other subtask outcomes)? Is it not logical, then, to want to run these independent tasks all at the same time? Such parallel processing could significantly improve the performance of the overall task. This is what MT programming is all about.

MT programming is ideal for programming tasks that are asynchronous in nature, require multiple concurrent activities, and where the processing of each activity may be *nondeterministic*, i.e., random and unpredictable. Such programming tasks can be organized or partitioned into multiple streams of execution where each has a specific task to accomplish. Depending on the application, these subtasks may calculate intermediate results that could be merged into a final piece of output.

While CPU-bound tasks may be fairly straightforward to divide into subtasks and executed sequentially or in a multithreaded manner, the task of managing a single-threaded process with multiple external sources of input is not as trivial. To achieve such a programming task without multithreading, a sequential program must use one or more timers and implement a multiplexing scheme.

A sequential program will need to sample each I/O (input/output) terminal channel to check for user input; however, it is important that the program does not block when reading the I/O terminal channel because the arrival of user input is nondeterministic, and blocking would prevent processing of other I/O channels. The sequential program must use non-blocked I/O or blocked I/O with a timer (so that blocking is only temporary).

Because the sequential program is a single thread of execution, it must juggle the multiple tasks that it needs to perform, making sure that it does not spend too much time on any one task, and it must ensure that user response time is appropriately distributed. The use of a sequential program for this type of programming task often results in a complicated flow of control program that is difficult to understand and maintain.

Using an MT program with a shared data structure such as a Queue (a multithreaded queue data structure discussed later in this chapter), this programming task can be organized with a few threads that have specific functions to perform:

1. **UserRequestThread:** responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.

2. **RequestProcessor:** a thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.

3. **ReplyThread:** responsible for taking output destined for the user and either sending it back, if in a networked application, or writing data to the local file system or database.

Organizing this programming task with multiple threads reduces the complexity of the program and enables an implementation that is clean, efficient, and well-organized. The logic in each thread is typically less complex because it has a specific job to do. For example, the **UserRequestThread** simply reads input from a user and places the data into a queue for further processing by another thread, etc. Each thread has its own job to do; and you merely have to design each type of thread to do one thing and do it well. Use of threads for specific tasks is not unlike Henry Ford's assembly line model for manufacturing automobiles.

Threads and Processes

What Are Processes?

Computer *programs* are merely executables, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A *process* (sometimes called a *heavyweight process*) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes. Processes can also *fork* or *spawn* new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

What Are Threads?

Threads (sometimes called *lightweight processes*) are similar to processes except that they all execute within the same process, thus all share the same context. They can be thought of as "mini-processes" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be pre-empted (interrupted) and temporarily put on hold (also known as *sleeping*) while other threads are running—this is called *yielding*.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial "back-of-the-line" to await getting more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a *race condition*. Fortunately, most thread libraries come with some sort of synchronization primitives which allow the thread manager to control execution and access.

Another caveat is that threads may not be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.

Threads and Python

Global Interpreter Lock

Execution by Python code is controlled by the *Python Virtual Machine* (a.k.a. the interpreter main loop), and Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only *one* is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by a *global interpreter lock* (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

- Set the GIL,
- Switch in a thread to run,
- Execute for a specified number of bytecode instructions,
- Put the thread back to sleep (switch out thread),
- Unlock the GIL, and,
- Do it all over again (rinse, lather, repeat).

When a call is made to external code, i.e., any C/C++ extension built-in function, the GIL will be locked until it has completed (since there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL however, so you being the Python developer shouldn't have to worry about your Python code locking up in those situations.

As an example, for any Python I/O-oriented routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code which *doesn't* have much I/O will tend to keep the processor (and GIL) to the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

Those of you interested in the source code, the interpreter main loop, and the GIL can take a look at `eval_code2()` routine in the `Python/ceval.c` file, which is the Python Virtual Machine.

Exiting Threads

When a thread completes execution of the function they were created for, they exit. Threads may also quit by calling an exit function such as `thread.exit()`, or any of the standard ways of exiting a Python process, i.e., `sys.exit()` or raising the `SystemExit` exception.

There are a variety of ways of managing thread termination. In most systems, when the main thread exits, all other threads die without cleanup, but for some systems, they live on. Check your operating system threaded programming documentation regarding their behavior in such occasions.

Main threads should always be good managers, though, and perform the task of knowing what needs to be executed by individual threads, what data or arguments each of the spawned threads requires, when they complete execution, and what results they provide. In so doing, those main threads can collate the individual results into a final conclusion.

Accessing Threads From Python

Python supports multithreaded programming, depending on the operating system that it is running on. It is supported on most versions of Unix, including Solaris and Linux, and Windows. Threads are not currently available on the Macintosh platform. Python uses POSIX-compliant threads, or "pthreads," as they commonly known.

By default, threads are not enabled when building Python from source, but are available for Windows platforms automatically from the installer. To tell whether threads are installed, simply attempt to import the thread module from the interactive interpreter. No errors occur when threads are available:


```
>>> import thread
>>>
```

If your Python interpreter was not compiled with threads enabled, the module import fails:

```
>>> import thread
Traceback (innermost last): File "<stdin>", line 1, in ?
ImportError: No module named thread
```

In such cases, you may have to recompile your Python interpreter to get access to threads. This usually involves invoking the configure script with the "--with-thread" option. Check the README file for your distribution for specific instructions on how to compile Python with threads for your system.

Due to the brevity of this chapter, we will give you only a quick introduction to threads and MT programming in Python. We refer you to the official documentation to get the full coverage of all the aspects of the threading support which Python has to offer. Also, we recommended accessing any general operating system textbook for more details on processes, interprocess communication, multi-threaded programming, and thread/process synchronization.

Life Without Threads

For our first set of examples, we are going to use the `time.sleep()` function to show how threads work. `time.sleep()` takes a floating point argument and "sleeps" for the given number of seconds, meaning that execution is temporarily halted for the amount of time specified.

Let us create two "time loops," one which sleeps for 4 seconds and one that sleeps for 2 seconds, `loop0()` and `loop1()`, respectively. (We use the names "loop0" and "loop1" as a hint that we will eventually have a sequence of loops.) If we were to execute `loop0()` and `loop1()` sequentially in a one-process or single-threaded program, as `onethr.py` does in [Example 17.1](#), the total execution time would be at least 6 seconds. There may or may not be a 1-second gap between the starting of `loop0()` and `loop1()`, and other execution overhead which may cause the overall time to be bumped to 7 seconds.

Example 17.1. Loops Executed by a Single Thread (`onethr.py`)

Executes two loops consecutively in a single-threaded program. One loop must complete before the other can begin. The total elapsed time is the sum of times taken by each loop.

```
<$nopcode>
001 1      #!/usr/bin/env python
002 2
003 3      from time import sleep, time, ctime
004 4
005 5      def loop0():
006 6          print 'start loop 0 at:', ctime(time())
007 7          sleep(4)
008 8          print 'loop 0 done at:', ctime(time())
009 9
010 10     def loop1():
011 11         print 'start loop 1 at:', ctime(time())
012 12         sleep(2)
```

```
13     13         print 'loop 1 done at:', ctime(time())
14     14
15     15 def main():
16     16         print 'starting...'
017 17         loop0()
018 18         loop1()
019 19         print 'all DONE at:', ctime(time())
020 20
21     21 if name == '_main_':
22     22         main()
23     <$nopcode>
```



We can verify this by executing `onethr.py`, which gives the following output:

```
% onethr.py starting...
start loop 0 at: Sun Aug 13 05:03:34 2000
loop 0 done at: Sun Aug 13 05:03:38 2000
start loop 1 at: Sun Aug 13 05:03:38 2000
loop 1 done at: Sun Aug 13 05:03:40 2000
all DONE at: Sun Aug 13 05:03:40 2000
```

Now, pretend that rather than sleeping, `loop0()` and `loop1()` were separate functions that performed individual and independent computations, all working to arrive at a common solution. Wouldn't it be useful to have them run in parallel to cut down on the overall running time? That is the premise behind MT that we will now introduce you to.

Python Threading Modules

Python provides several modules to support MT programming, including the `thread`, `threading`, and `Queue` modules. The `thread` and `threading` modules allow the programmer to create and manage threads. The `thread` module provides the basic thread and locking support, while `threading` provides high-level full-featured thread management. The `Queue` module allows the user to create a queue data structure which can be shared across multiple threads. We will take a look at these modules individually, present a good number of examples, and a couple of intermediate-sized applications.

threadModule

Let's take a look at what the `thread` module has to offer. In addition to being able to spawn threads, the `thread` module also provides a basic synchronization data structure called a *lock object* (a.k.a. primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore). As we mentioned earlier, such synchronization primitives go hand-in-hand with thread management.

Listed in [Table 17.1](#) are a list of the more commonly-used thread functions and `LockType` lock object methods:

<i>Function/Method</i>	<i>Description</i>
threadModule Functions	
<code>start_new_thread(function, args, kwargs=None)</code>	spawns a new thread and execute <i>function</i> with the given <i>args</i> and optional <i>kwargs</i>
<code>allocate_lock()</code>	allocates <code>LockType</code> lock object
<code>exit()</code>	instructs a thread to exit
LockType Lock Object Methods	
<code>acquire(wait=None)</code>	attempts to acquire lock object
<code>locked()</code>	returns 1 if lock acquired, 0 otherwise
<code>release()</code>	releases lock

Table 17.1. threadModule and Lock Objects

The key function of the `thread` module is `start_new_thread()`. Its syntax is exactly that of the `apply()` built-in function, taking a function along with arguments and optional keyword arguments. The difference is that instead of the main thread executing the function, a new thread is spawned to invoke the function.

Let's take our `onethr.py` example and integrate threading into it. By slightly changing the call to the `loop*()` functions, we now present `mtsleep1.py` in [Example 17.2](#).

Example 17.2. Using the threadModule (mtsleepl.py)

The same loops from onethr.py are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

<\$nopcode>

```
001 1      #!/usr/bin/env python
002 2
003 3  import thread
004 4  from time import sleep, time, ctime
005 5
006 6  def loop0():
007 7      print 'start loop 0 at:', ctime(time())
008 8      sleep(4)
009 9      print 'loop 0 done at:', ctime(time())
010 10
011 11 def loop1():
012 12     print 'start loop 1 at:', ctime(time())
013 13     sleep(2)
014 14     print 'loop 1 done at:', ctime(time())
015 15
016 16 def main():
017 17     print 'starting threads...'
018 18     thread.start_new_thread(loop0, ())
019 19     thread.start_new_thread(loop1, ())
020 20     sleep(6)
021 21     print 'all DONE at:', ctime(time())
022 22
023 23 if __name__ == '__main__':
024 24     main()
025     <$nopcode>
```



start_new_thread() requires the first two arguments, so that's the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4, the length of time of our longest loop, plus any overhead.

```
% mtsleep1.py starting threads...
start loop 0 at: Sun Aug 13 05:04:50 2000
start loop 1 at: Sun Aug 13 05:04:50 2000
loop 1 done at: Sun Aug 13 05:04:52 2000
loop 0 done at: Sun Aug 13 05:04:54 2000
all DONE at: Sun Aug 13 05:04:56 2000
```

The pieces of code that sleep for 4 and 2 seconds now occur concurrently, contributing to the lower overall runtime.

The only other major change to our application is the addition of the "sleep(6)" call. Why is this necessary? The reason is that if we did not stop the main thread from continuing, it would proceed to the next statement, displaying "all done" and exit, killing both threads running loop0() and loop1().

We did not have any code which told the main thread to wait for the child threads to complete before continuing. This is what we mean by threads requiring some sort of synchronization. In our case, we used another sleep() call as our synchronization mechanism. We used a value of 6 seconds because we know that both threads (which take 4 and 2 seconds, as you know) should have completed by the time the main thread has counted to 6.

You are probably thinking that there should be a better way of managing threads than creating that extra delay of 6 seconds in the main thread. Because of this delay, the overall runtime is no better than in our single-threaded version. Using sleep() for thread synchronization as we did is not reliable. What if our loops had independent and varying execution times? We may be exiting the main thread too early or too late. This is where locks come in.

Making yet another update to our code to include locks as well as getting rid of separate loop functions, we get mtsleep2.py, presented in [Example 17.3](#). Running it, we see that the output is similar to mtsleep1.py. The only difference is that we did not have to wait the extra time for mtsleep1.py to conclude. By using locks, we were able to exit as soon as both threads had completed execution.

```
% mtsleep2.py starting threads...
start loop 0 at: Sun Aug 13 16:34:41 2000
start loop 1 at: Sun Aug 13 16:34:41 2000
loop 1 done at: Sun Aug 13 16:34:43 2000
loop 0 done at: Sun Aug 13 16:34:45 2000
all DONE at: Sun Aug 13 16:34:45 2000
```

Example 17.3. Using thread and Locks (mtsleep2.py)

Rather than using a call to sleep() to hold up the main thread as in mtsleep1.py, the use of locks makes more sense.

```
<$nopcode>
001 1      #!/usr/bin/env python
002 2
003 3      import thread
004 4      from time import sleep, time, ctime 005 5
006 6      loops = [ 4, 2 ]
007 7
008 8      def loop(nloop, nsec, lock):
009 9          print 'start loop', nloop, 'at:', ctime(time())
010 10         sleep(nsec)
011 11         print 'loop', nloop, 'done at:', ctime(time())
```

```
012 12         lock.release()
013 13
14 14 def main():
15 15     print 'starting threads...'
16 16     locks = []
17 17     nloops = range(len(loops))
018 18
19 19     for i in nloops:
20 20         lock = thread.allocate_lock()
21 21         lock.acquire()
22 22         locks.append(lock)
023 23
24 24     for i in nloops:
```



```

25 25          thread.start_new_thread(loop, \
26 26          (i, loops[i], locks[i]))
027 27
28 28          for i in nloops:
29 29              while locks[i].locked(): pass <$nopcode>
030 30
031 31          print 'all DONE at:', ctime(time()) 032 32
33 33 if name      == '_main_':
34 34     main()
35     <$nopcode>

```

So how did we accomplish our task with locks? Let's take a look at the source code:

Line-by-line explanation

Lines 1–6

After the Unix start-up line, we import the thread module and a few familiar attributes of the time module. Rather than hardcoding separate functions to count to 4 and 2 seconds, we will use a single loop() function and place these constants in a list, loops.

Lines 8–12

The loop() function will proxy for the now-removed loop*() functions from our earlier examples. We had to make some cosmetic changes to loop() so that it can now perform its duties using locks. The obvious changes are that we need to be told which loop number we are as well as how long to sleep for. The last piece of new information is the lock itself. Each thread will be allocated an acquired lock. When the sleep() time has concluded, we will release the corresponding lock, indicating to the main thread that this thread has completed.

Lines 14–34

The bulk of the work is done here in main() using three separate **for** loops. We first create a list of locks, which we obtain using the thread.allocate_lock() function and acquire each lock with the acquire() method. Acquiring a lock has the effect of "locking the lock." Once it's locked, we add the lock to the lock list, locks. The next loop actually spawns the threads, invoking the loop() function per thread, and for each thread, provides it with the loop number, the time to sleep for, and the acquired lock for that thread. So why didn't we start the threads in the lock acquisition loop? There are several reasons: (1) we wanted to synchronize the threads, so that "all the horses started out the gate" around the same time, and (2) locks take a little bit of time to be acquired. If your thread executes "too fast," it is possible that it completes before the lock has a chance to be acquired.

It is up to each thread to unlock its lock object when it has completed execution. The final loop just sits-and-spins (pausing the main thread) until both locks have been released before continuing execution. Since we are checking each lock sequentially, we may be at the mercy of all the slower loops if they are more towards the beginning of the set of locks. In such cases, the majority of the wait time may be for the first loop(s). When that lock is released, remaining locks may have already been unlocked (meaning that corresponding threads have completed execution). The result is that the main thread will fly through those lock checks without pause. Finally, you should be well aware that the final pair of lines will execute main() only if we are invoking this script directly.

As hinted in the earlier Core Note, we presented the thread module only to introduce the reader to threaded programming. Your MT application should use higher-level modules such as the threading module.

threadingModule

We will now introduce the higher-level threading module which gives you not only a Thread class but also a wide variety of synchronization mechanisms to use to your heart's content. [Table 17.2](#) represents a list of all the objects which are provided for in the threadingmodule.

threading Module Objects	Description
Thread	object which represents a single thread of execution
Lock	primitive lock object (same lock object as in the thread module)
RLock	re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
Condition	condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value
Event	general version of condition variables whereby any number of threads are waiting for some event to occur and all will awaken when the event happens
Semaphore	provides a "waiting area"-like structure for threads waiting on a lock

Table 17.2. threadingModule Objects

In this section, we will examine how to use the Thread class to implement threading. Since we have already covered the basics of locking, we will not cover the locking primitives here. The Thread() class also contains a form of synchronization, so explicit use of locking primitives is not necessary.

ThreadClass

There are a variety of ways you can create threads using the Thread class. We cover three of them here, all quite similar. Pick the one you feel most comfortable with, not to mention the most appropriate for your application and future scalability (we like choice 3 the best):

1. Create Threadinstance, passing in function
2. Create Threadinstance, passing in callable class instance
3. Subclass Thread and create subclass instance

Create Threadinstance, passing in function

In our first example, we will just instantiate Thread, passing in our function (and its arguments) in a manner similar to our previous examples. This function is what will be executed when we direct the thread to begin execution. Taking our mtsleep2.py script and tweaking it, adding the use of Thread objects, we have mtsleep3.py, shown in [Example 17.4](#).

When we run it, we see output similar to its predecessors':

```
% mtsleep3.py starting threads...
start loop 0 at: Sun Aug 13 18:16:38 2000
start loop 1 at: Sun Aug 13 18:16:38 2000
loop 1 done at: Sun Aug 13 18:16:40 2000
loop 0 done at: Sun Aug 13 18:16:42 2000
all DONE at: Sun Aug 13 18:16:42 2000
```

So what *did* change? Gone are the locks which we had to implement when using the thread module. Instead, we create a set of Thread objects. When each Thread is instantiated, we dutifully pass in the function (target) and arguments (args) and receive a Thread instance in return. The biggest difference between instantiating Thread [calling Thread()] and invoking thread.start_new_thread() is that the new thread does not begin execution right away. This is a useful synchronization feature, especially when you don't want the threads to start immediately.

Example 17.4. Using the threading Module (mthreads.py)

The Thread class from the threading module has a join() method which lets the main thread wait for thread completion.

<\$nopage>

```
001 1      #!/usr/bin/env python
002 2
003 3      import threading
004 4      from time import sleep, time, ctime 005 5
006 6      loops = [ 4, 2 ]
007 7
008 8      def loop(nloop, nsec):
009 9          print 'start loop', nloop, 'at:', ctime(time())
010 10         sleep(nsec)
011 11         print 'loop', nloop, 'done at:', ctime(time())
012 12
013 13 def main():
014 14     print 'starting threads...'
015 15     threads = []
016 16     nloops = range(len(loops))
017 17
018 18     for i in nloops:
019 19         t = threading.Thread(target=loop,
020 20             args=(i, loops[i]))
021 21         threads.append(t)
022 22
023 23     for i in nloops:                 # start threads
024 24         threads[i].start()
025 25
026 26     for i in nloops:                 # wait for all
027 27         threads[i].join()           # threads to finish
028 28
029 29     print 'all DONE at:', ctime(time()) 030 30
031 31 if name == '_main_':
032 32     main()
033 33 <$nopage>
```

Once all the threads have been allocated, we let them go off to the races by invoking each thread's `start()` method, but not a moment before that. And rather than having to manage a set of locks (allocating, acquiring, releasing, checking lock state, etc.), we simply call the `join()` method for each thread. `join()` will wait until a thread terminates, or, if provided, a timeout occurs. Use of `join()` appears much cleaner than an infinite loop waiting for locks to be released (causing these locks to sometimes be known as "spin locks").

One other important aspect of `join()` is that it does not need to be called at all. Once threads are started, they will execute until their given function completes, whereby they will exit. If your main thread has things to do other than wait for threads to complete (such as other processing or waiting for new client requests), it should be all means do so. `join()` is useful only when you *want* to wait for thread completion.

Create Threadinstance, passing in callable class instance

A similar offshoot to passing in a function when creating a thread is to have a callable class and passing in an instance for execution—this is the more OO approach to MT programming. Such a callable class embodies an execution environment that is much more flexible than a function or choosing from a set of functions. You now have the power of a class object behind you, as opposed to a single function or a list/tuple of functions.

Adding our new class `ThreadFunc` to the code and making other slight modifications to `mtsleep3.py`, we get `mtsleep4.py`, given in [Example 17.5](#). If we run `mtsleep4.py`, we get the expected output:

```
% mtsleep4.py starting threads...
start loop 0 at: Sun Aug 13 18:49:17 2000
start loop 1 at: Sun Aug 13 18:49:17 2000
loop 1 done at: Sun Aug 13 18:49:19 2000
loop 0 done at: Sun Aug 13 18:49:21 2000
all DONE at: Sun Aug 13 18:49:21 2000
```

So what are the changes this time? The addition of the `ThreadFunc` class and a minor change to instantiate the `Thread` object, which also instantiates `ThreadFunc`, our callable class. In effect, we have a double instantiation going on here. Let's take a closer look at our `ThreadFunc` class.

We want to make this class general enough to use with other functions besides our `loop()` function, so we added some new infrastructure, such as having this class hold the arguments for the function, the function itself, and also a function name string. The constructor `__init__()` just sets all the values.

Example 17.5. Using Callable classes (`mtsleep4.py`)

In this example we pass in a callable class (instance) as opposed to just a function. It presents more of an OO approach than `mtsleep3.py`.

```
<$nopage>
001 1  #!/usr/bin/env python
002 2
003 3  import threading
004 4  from time import sleep, time, ctime
005 5
006 6  loops = [ 4, 2 ]
007 7
008 8  class ThreadFunc:
009 9
010 10 def __init__(self, func, args, name=""):
011 11     self.name = name
012 12     self.func = func
013 13     self.args = args
```

```

014 14
015 15 def _call_(self):
016 16     apply(self.func, self.args)
017 17
018 18 def loop(nloop, nsec):
019 19     print 'start loop', nloop, 'at:', ctime(time())
020 20     sleep(nsec)
021 21     print 'loop', nloop, 'done at:', ctime(time())
022 22
023 23 def main():
024 24     print 'starting threads...'
025 25     threads = []
026 26     nloops = range(len(loops))
027 27
028 28     for i in nloops: # create all threads
029 29         t = threading.Thread(\
030 30             target=ThreadFunc(loop, (i, loops[i]),
031 31             loop_name_))
032 32         threads.append(t)
033 33
034 34     for i in nloops: # start all threads
035 35         threads[i].start()
036 36
037 37     for i in nloops: # wait for completion
038 38         threads[i].join()
039 39
040 40     print 'all DONE at:', ctime(time())
041 41
042 42 if _name == '_main_':
043 43     main()
044 <$npage>

```

When the Thread code calls our ThreadFunc object when a new thread is created, it will invoke the `_call_()` special method. Because we already have our set of arguments, we do not need to pass it to the `Thread()` constructor, but do have to use `apply()` in our code now because we have an argument tuple.

```
self.res = self.func(*self.args)
```

Subclass Thread and create subclass instance

The final introductory example involves subclassing `Thread()`, which turns out to be extremely similar to creating a callable class as in the previous example. Subclassing is a bit easier to read when you are creating your threads (lines 28–29). We will present the code for `mtsleep5.py` in [Example 17.6](#) as well as the output obtained from its execution, and leave it as an exercise for the reader to compare `mtsleep5.py` to `mtsleep4.py`.

Here is the output for `mtsleep5.py`, again, just what we expected:

```

% mtsleep5.py starting threads...
start loop 0 at: Sun Aug 13 19:14:26 2000
start loop 1 at: Sun Aug 13 19:14:26 2000
loop 1 done at: Sun Aug 13 19:14:28 2000
loop 0 done at: Sun Aug 13 19:14:30 2000

```

all DONE at: Sun Aug 13 19:14:30 2000

While the reader compares the source between the mtsleep4 and mtsleep5 modules, we want to point out the most significant changes: (1) our MyThread subclass constructor must first invoke the base class constructor (line 9), and (2) the former special method

`_call_()` must be called `run()` in the subclass.

We now modify our MyThread class with some diagnostic output and store it in a separate module called `myThread` (see [Example 17.7](#)) and import this class for the upcoming examples. Rather than simply calling `apply()` to run our functions, we also save the result to instance attribute `self.res`, and create a new method to retrieve that value, `getResult()`.

Example 17.6. Subclassing Thread(mtsleep5.py)

Rather than instantiating the Thread class, we subclass it. This gives us more flexibility in customizing our threading objects and simplifies the thread creation call.

<\$nopage>

```
001 1      #!/usr/bin/env python
002 2
003 3      import threading
004 4      from time import sleep, time, ctime
005 5
006 6      loops = ( 4, 2 )
007 7
008 8      class MyThread(threading.Thread):
009 9          def init (self, func, args, name=""):
010 10             threading.Thread. init (self)
011 11             self.name = name
012 12             self.func = func
013 13             self.args = args
014 14
015 15         def run(self):
016 16             apply(self.func, self.args)
017 17
018 18 def loop(nloop, nsec):
019 19     print 'start loop', nloop, 'at:', ctime(time())
020 20     sleep(nsec)
021 21     print 'loop', nloop, 'done at:', ctime(time())
022 22
023 23 def main():
024 24     print 'starting threads...'
025 25     threads = []
026 26     nloops = range(len(loops))
027 27
028 28     for i in nloops:
029 29         t = MyThread(loop, (i, loops[i]), \
030 30             loop. name )
031 31         threads.append(t)
032 32
033 33     for i in nloops:
```



```
34     34         threads[i].start()
035 35
36     36         for i in nloops:
37     37             threads[i].join()
038 38
039 39         print 'all DONE at:', ctime(time())
040 40
41     41 if name == '_main_':
42     42         main()
43     <$nopage>
```



Example 17.7. MyThreadSubclass of Thread (myThread.py)

To generalize our subclass of Thread from mtsleep5.py, we move the subclass to a separate module and add a getResult() method for callables which produce return values.

```
<$nopage>
001 1      #!/usr/bin/env python
002 2
03   3  import threading
04   4  from time import time, ctime
05   005 5
06   6  class MyThread(threading.Thread):
07   7      def init (self, func, args, name=""):
08   008 8          threading.Thread. init (self)
09   9          self.name = name
10  10          self.func = func
11  11          self.args = args
12  12
13  13  def      getResult(self):
14  14          return self.res
15  15
16  16  def      run(self):
17  17          print 'starting', self.name, 'at:', \
18  18          ctime(time())
19  19          self.res = apply(self.func, self.args)
20  20          print self.name, 'finished at:', \
21  21          ctime(time())
22  <$nopage>
```

Fibonacci and factorial... take 2, plus summation

The mtfacfib.py script, given in [Example 17.8](#), compares execution of the recursive Fibonacci, factorial, and summation functions. This script runs all three functions in a single-threaded manner, then performs the same task using threads to illustrate one of the advantages of having a threading environment.

Example 17.8. Fibonacci, Factorial, Summation (mtfacfib.py)

In this MT application, we execute 3 separate recursive functions—first in a single-threaded fashion, followed by the alternative with multiple threads.

```
<$nopage>
001 1      #!/usr/bin/env python
002 2
03   3  from myThread import MyThread
04   4  from time import time, ctime, sleep
05   5
06  6      def fib(x):
07  7          sleep(0.005)
08  8          if x < 2: return 1
09  9          return (fib(x-2) + fib(x-1))
10  10
11  11 def fac(x):
```

```

012 12         sleep(0.1)
13   13         if x < 2: return 1
14   14         return (x * fac(x-1))
015 15
016 16 def sum(x):
017 17         sleep(0.1)
18   18         if x < 2: return 1
19   19         return (x + sum(x-1))
020 20
021 21 funcs = [fib, fac, sum] 022 22 n = 12
023 23
24   24 def main():
25   25         nfuncs = range(len(funcs))
026 26
27   27         print '*** SINGLE THREAD'
28   28         for i in nfuncs:
29   29                 print 'starting', funcs[i]. name , 'at:', \
030 30 ctime(time())
031 31         print funcs[i](n)
032 32         print funcs[i]. name , 'finished' at:', \
033 33 ctime(time())
034 34
035 35         print '\n*** MULTIPLE THREADS'
036 36         threads = []
037 37         for i in nfuncs:
038 38                 t = MyThread(funcs[i], (n,),
039 39                 funcs[i]. name )
040 40                 threads.append(t)
041 41
042 42         for i in nfuncs:
043 43                 threads[i].start()
044 44
045 45         for i in nfuncs:
046 46                 threads[i].join()
047 47         print threads[i].getResult()
048 48
049 49         print 'all DONE'
050 50
051 51         if _name == ' _main ':
052 52                 main()
053 <$nopage>

```

Running in single-threaded mode simply involves calling the functions one at a time and displaying the corresponding the results right after the function call.

When running in multithreaded mode, we do not display the result right away. Because we want to keep our MyThread class as general as possible (being able to execute callables which do and do not produce output), we wait until the end to call the getResult() method to finally show you the return values of each function call.

Because these functions execute so quickly (well, maybe except for the Fibonacci function), you will notice that we had to add calls to `sleep()` to each function to slow things down so that we can see how threading may improve performance, if indeed the actual work had varying execution times—you certainly wouldn't pad your work with calls to `sleep()`. Anyway, here is the output:

```
% mtfacfib.py
*** SINGLE THREAD
starting fib      at:  Sun Jun 18 19:52:20    2000
233
fib finished     at:  Sun Jun 18 19:52:24    2000
starting fac     at:  Sun Jun 18 19:52:24    2000
479001600
fac finished     at:  Sun Jun 18 19:52:26    2000
starting sum     at:  Sun Jun 18 19:52:26    2000
78
sum finished     at:  Sun Jun 18 19:52:27    2000
*** MULTIPLE THREADS
starting fib at: Sun Jun 18 19:52:27 2000
starting fac at: Sun Jun 18 19:52:27 2000
starting sum at: Sun Jun 18 19:52:27 2000
fac finished     at:  Sun Jun 18 19:52:28    2000
sum finished     at:  Sun Jun 18 19:52:28    2000
fib finished     at:  Sun Jun 18 19:52:31    2000
233
479001600
78
all DONE
```

Producer-Consumer Problem and the QueueModule

The final example illustrates the producer-consumer scenario where a producer of goods or services creates goods and places it in a data structure such as a queue. The amount of time between producing goods is non-deterministic, as is the consumer consuming the goods produced by the producer.

We use the Queue module to provide an interthread communication mechanism which allows threads to share data with each other. In particular, we create a queue for the producer (thread) to place new goods into and where the consumer (thread) can consume goods from.

In particular, we will use the following attributes from the Queue module (see [Table 17.3](#)).

<i>Function/Method</i>	<i>Description</i>
Queue Module Function	
<code>queue(size)</code>	creates a Queue object of given <i>size</i>
Queue Object Methods	
<code>qsize()</code>	returns queue size (approximate, since queue may be getting updated by other threads)
<code>empty()</code>	returns 1 if queue empty, 0 otherwise
<code>full()</code>	returns 1 if queue full, 0 otherwise
<code>put(item, block=0)</code>	puts <i>item</i> in queue, if <i>block</i> given (not 0), block until room is available
<code>get(block=0)</code>	gets <i>item</i> from queue, if <i>block</i> given (not 0), block until an item is available

Table 17.3. Common QueueModule Attributes

Example 17.9. Producer-Consumer Problem (prodcons.py)

We feature an implementation of the Producer–Consumer problem using Queue objects and a random number of goods produced (and consumed). The producer and consumer are individually—and concurrently—executing threads.

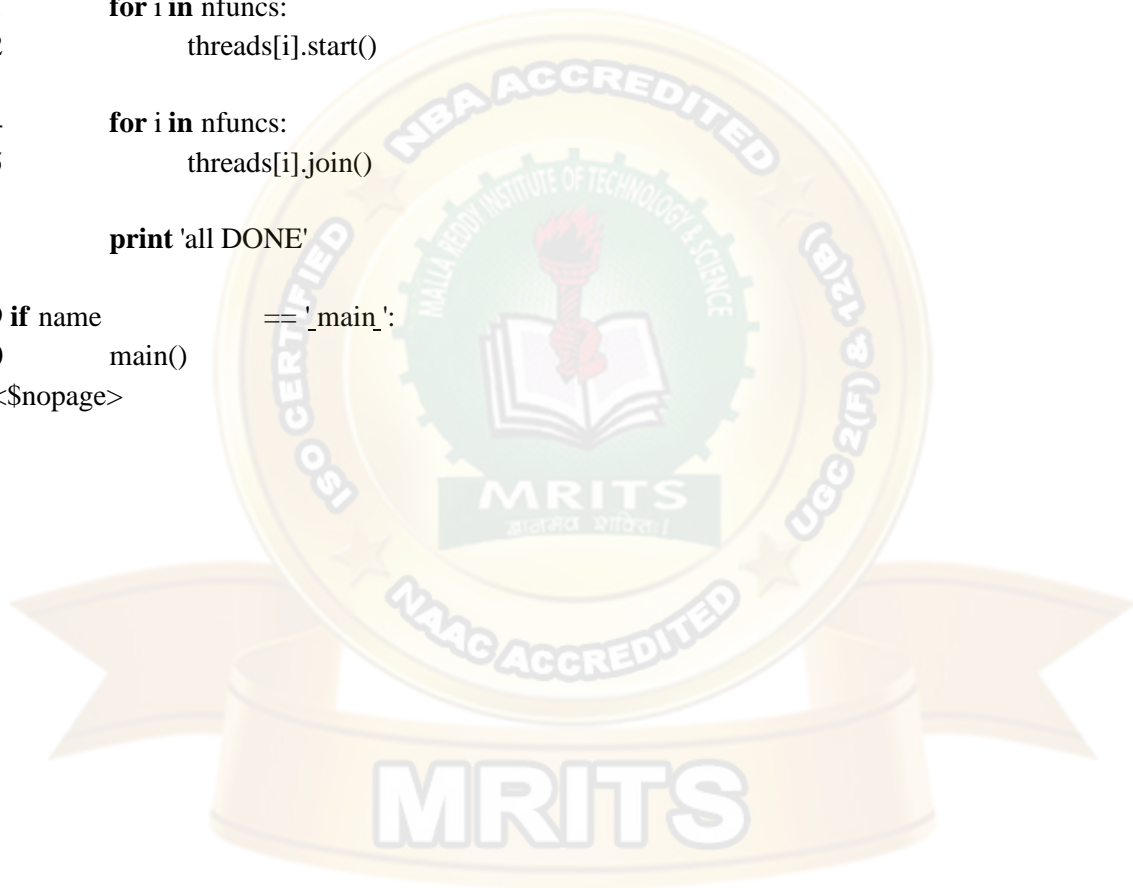
<\$nopage>

```

001 1      #!/usr/bin/env python
002 2
003 3      from random import randint
004 4      from time import time, ctime, sleep
005 5      from Queue import Queue
006 6      from myThread import MyThread
007 7
008 8      def writeQ(queue):
009 9          print 'producing object for Q...',
010 10         queue.put('xxx', 1)
011 11         print "size now", queue.qsize()
012 12
013 13     def readQ(queue):
014 14         val = queue.get(1)
015 15     print 'consumed object from Q... size now', \
016 16         queue.qsize()
017 17
018 18     def writer(queue, loops):
019 19         for i in range(loops):
020 20             writeQ(queue)
021 21             sleep(randint(1, 3))
022 22
023 23     def reader(queue, loops):
024 24         for i in range(loops):
025 25             readQ(queue)
026 26             sleep(randint(2, 5))
027 27
028 28     funcs = [writer, reader]
029 29     nfuncs = range(len(funcs))

```

```
030 30
31     31 def main():
32     32         nloops = randint(2, 5)
033 33         q = Queue(32)
034 34
35     35         threads = []
36     36         for i in nfuncs:
37     37             t = MyThread(funcs[i], (q, nloops), \
38     038 38                 funcs[i]. name )
039 39             threads.append(t)
040 40
41     41         for i in nfuncs:
42     42             threads[i].start()
043 43
44     44         for i in nfuncs:
45     45             threads[i].join()
046 46
047 47         print 'all DONE'
048 48
49     49 if name == '_main_':
50     50         main()
51     <$nopcode>
```



Here is the output from one execution of this script:

```
% prodcons.py
starting writer at: Sun Jun 18 20:27:07 2000 producing object for Q... size now 1
starting reader at: Sun Jun 18 20:27:07 2000 consumed object from Q... size now 0
producing object for Q... size now 2 producing object for Q... size now 3 consumed object from Q...
size now 2 consumed object from Q... size now 1
writer finished at: Sun Jun 18 20:27:17 2000 consumed object from Q... size now 0
reader finished at: Sun Jun 18 20:27:25 2000 all DONE
As you can see, the producer and consumer do not necessarily alternate in execution. (Thank goodness
for random numbers!) Seriously though, real life is generally random and non-deterministic.
producing object for Q... size now 1 consumed object from Q... size now 0 producing object for Q...
size now 1
```

Line-by-line explanation Lines 1–6

In this module, we will use the `Queue.Queue` object as well as our thread class `myThread.MyThread` which we gave in [Example 17.7](#). We will use `random.randint()` to make production and consumption somewhat varied, and also grab the usual suspects from the `time` module.

Lines 8–16

The `writeQ()` and `readQ()` functions each have a specific purpose, to place an object in the queue—we are using the string 'xxx' for example—and to consume a queued object, respectively. Notice that we are producing one object and reading one object each time.

Lines 18–26

The `writer()` is going to run as a single thread whose sole purpose is to produce an item for the queue, wait for a bit, then do it again, up to the specified number of times, chosen randomly per script execution. The `reader()` will do likewise, with the exception of consuming an item, of course.

You will notice that the random number of seconds that the writer sleeps is in general shorter than the amount of time the reader sleeps. This is to discourage the reader from trying to take items from an empty queue. By giving the writer a shorter time period of waiting, it is more likely that there will already be an object for the reader to consume by the time their turn rolls around again.

Lines 28–29

These are just setup lines to set the total number of threads that are to be spawned and executed.

Lines 31–47

Finally, our `main()` function, which should look quite similar to the `main()` in all of the other scripts in this chapter. We create the appropriate threads and send them on their way, finishing up when both threads have concluded execution.

We infer from this example that a program that has multiple tasks to perform can be organized to use separate threads for each of the tasks. This can result in a much cleaner program design than a single threaded program that attempts to do all of the tasks.

UNIT - IV

GUI Programming: Introduction, Tkinter and Python Programming, Brief Tour of Other GUIs, Related Modules and Other GUIs

WEB Programming: Introduction, Web Surfing with Python, Creating Simple Web Clients, Advanced Web Clients, CGI-Helping Servers Process Client Data, Building CGI Application Advanced CGI, Web (HTTP) Servers

Explain in detail about GUI Programming with Tkinter.

GUI Programming with Tkinter

Introduction

What Are Tcl, Tk, and Tkinter?

Tkinter is Python's default graphical user interface library. It is based on the Tk toolkit, originally designed for the Tool Command Language (TCL). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk) and Python (Tkinter). With the GUI development portability and flexibility of Tk, along with the simplicity of scripting language integrated with the power of systems language, you are given the tools to rapidly design and implement a wide variety of commercial-quality GUI applications.

If you are new to GUI programming, you will be pleasantly surprised at how easy it is. You will also find that Python, along with Tkinter, provide a fast and exciting way to build applications that are fun (and perhaps useful) and that would have taken much longer if you had to program directly in C/C++ with the native windowing system's libraries. Once you have designed the application and the look-and-feel that goes along with your program, you will use basic building blocks known as widgets to piece together the desired components, and finally, to attach functionality to "make it real."

If you are an "old-hat" at using Tk, either with Tcl or Perl, you will find Python a refreshing way to program GUIs, on top of that, it provides an even faster rapid prototyping system for building GUIs. Remember that you also have Python's system-accessibility, networking functionality, XML, numerical and visual processing, database access, and all the other standard library and third-party extension modules.

Once you get Tkinter up on your system, it will take less than 15 minutes to get your first GUI app running.

Getting Tkinter Installed and Working

Like threading, Tkinter is not necessarily turned on by default on your system. You can tell whether Tkinter is available for your Python interpreter by attempting to import the Tkinter module. If Tkinter is available, then no errors occur:

```
>>> import Tkinter
>>>
```

If your Python interpreter was not compiled with Tkinter enabled, the module import fails:

```
>>> import Tkinter
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/usr/lib/python1.5/lib-tk/Tkinter.py", line 8, in ?
```

```
import _tkinter # If this fails your Python may not be configured for Tk
ImportError: No module named _tkinter
```

You may have to recompile your Python interpreter to get access to Tkinter. This usually involves editing the Modules/Setup file and enabling all the correct settings to compile your Python interpreter with hooks to Tkinter or choosing to have Tk installed on your system. Check the README file for your Python distribution for specific instructions on getting Tkinter to compile on your system. Be sure, after your compilation, that you start the *new* Python interpreter you just created; otherwise, it will act just like your old one without Tkinter (and in fact, it *is* your old one).

Client-Server Architecture—Take 2

In the earlier chapter on network programming, we introduced the notion of client-server computing. A windowing system is another example of a software server. These run on a machine with an attached display, such as a monitor of some sort. There are clients too—programs which require a windowing environment to execute, also known as GUI applications. Such applications cannot run without a windows system.

The architecture becomes even more interesting when networking comes into play. Usually when a GUI application is executed, it displays to the machine that it started on (via the windowing server), but it is possible in some networked windowing environments, such as the X Windows system on Unix, to choose another machine's window server to display to. In such situations, you can be running a GUI program on one machine, but have it displayed at another!

Tkinter and Python Programming

TkinterModule: Adding Tk to your Applications

So what do you need to do to have Tkinter as part of your application? Well, first of all, it is not necessary to have an application already. You can create a pure graphical user interface if you want, but it probably isn't too useful without some underlying software that does something interesting.

There are basically five main steps that are required to get your GUI up-and-running:

1. Import the Tkinter module (or **from Tkinter import ***).
2. Create a top-level windowing object which contains your entire GUI application.
3. Built all your GUI components (and functionality) on top (or "inside") of your top-level windowing object).
4. Connect these GUI components to the underlying application code.
5. Enter the main event loop.

The first step is trivial: All GUIs which use Tkinter must import the Tkinter module. Getting access to Tkinter is the first step (see the previous [Section 18.1.2](#)).

Introduction to GUI Programming

Before going to the examples, we will give you a brief introduction to GUI application development in general. This will give you some of the background you need to move forward.

Setting up a GUI application is similar to an artist's producing a painting. Conventionally, there is a single canvas onto which the artist must put all the work. The way it works is like this: You start with a clean slate, a "top-level" windowing object on which you build the rest of your components. Think of it as a foundation to a house or the easel for an artist. In other words, you have to pour the concrete or set up your easel before putting together the actual structure or canvas on top of it. In Tkinter, this foundation is known as the top-level window object.

In GUI programming, a top-level root windowing object contains all of the little windowing objects that will be part of your complete GUI application. These can be text labels, buttons, list boxes, etc. These individual little GUI components are known as *widgets*. So when we say create a top-level window, we just mean that you need such a thing as a place where you put all your widgets. In Python, this would typically look like the line below :

```
top = Tkinter.Tk() # or just Tk() with "from Tkinter import *"
```

The object returned by Tkinter.Tk() is usually referred to as the *root object*, hence the reason why some applications use root rather than top to indicate as such. Top-level windows are those which show up standalone as part of your application, and, yes, you may have more than one top-level window for your GUI, but only one of them should be your root window. You may choose to completely design all your widgets first, then add the real functionality, or do a little of this and a little of that along the way. (This means mixing and matching steps 3 and 4 from our list above.)

Widgets may be standalone or be containers. If a widget "contains" other widgets, it is considered the *parent* of those widgets. Accordingly, if a widget is "contained" in another widget, it's considered a *child* of the parent, the parent being the next immediate enclosing container widget.

Usually, widgets have some associated behaviors, such as when a button is pressed, or text is filled into a text field.

These types of user behaviors are called *events*, and the actions that the GUI takes to respond to such events are known as *callbacks*.

Actions may include the actual button press (and release), mouse movement, hitting the RETURN or Enter key, etc. All of these are known to the system literally as *events*. The entire system of events which occurs from the beginning to the end of a GUI application is what drives it. This is known as event-driven processing.

One example of an event with a callback is a simple mouse move. Let's say the mouse pointer is sitting somewhere on top of your GUI application. If the mouse is moved to another part of your application, something has to cause the movement of the mouse on your screen so that it *looks* as if it is moving to another location. These are mouse move events that the system must process to give you the illusion (and reality) that your mouse is moving across the window. When you release the mouse, there are no more events to process, so everything just sits there quietly on the screen again.

The event-driven processing nature of GUIs fits right in with client-server architecture. When you start a GUI application, it must perform some setup procedures to prepare for the core execution, just as when a network server has to allocate a socket and bind it to a local address. The GUI application must establish all the GUI components, then draw (a.k.a. render or paint) them to the screen. Tk has a couple of geometry managers which help position the widget in the right place; the main one which you will use is called the *packer*. Once the packer has determined the sizes and alignments of all your widgets, it will then place them on the screen for you.

When all of the widgets, including the top-level window finally appear on your screen, your GUI application then enters a "server-like" infinite loop. This infinite loop involves waiting for a GUI event, processing it, then going back to wait for the next event.

The final step we described above says to enter the main loop once all the widgets are ready. This is the "server" infinite loop we have been referring to. In Tkinter, the code that does this is:

```
Tkinter.mainloop()
```

This is normally the last piece of sequential code your program runs. When the main loop is entered, the GUI takes over execution from there. All other action is via callbacks, even exiting your application. When you pull down the File menu to click on the Exit menu option or close the window directly, a callback must be invoked to end your GUI application.

Top-level window: Tkinter.Tk()

We mentioned above that all main widgets are built into the top-level window object. This object is created by the Tk class in Tkinter and is created via the normal instantiation:

```
>>> import Tkinter
>>> top = Tkinter.Tk()
```

Within this window, you place individual widgets or multiple-component pieces together to form your GUI.

Tk Widgets

There are currently 15 types of widgets in Tk. We present these widgets as well as a brief description in [Table 18.1](#).

We won't go over the Tk widgets in detail as there is plenty of good documentation available on them, either from the Tkinter topics page at the main Python Web site or the abundant number of Tcl/Tk printed and online resources (some of which are available in the Appendix). However, we will present several simple examples to help you get started.

NOTE

GUI development really takes advantage of default arguments in Python because there are numerous default actions in Tkinter widgets. Unless you know every single option available to you for every single widget you are using, it's best to start out by setting only the parameters you are aware of and letting the system handle the rest. These defaults were chosen carefully. If you do not provide these values, do not worry about your applications appearing odd on the screen. They were created with an optimized set of default arguments as a general rule, and only when you know how to exactly customize your widgets should you use values other than the default.

Table 18.1. Tk Widgets

<i>Widgets</i>	<i>Description</i>
Button	similar to a Label but provides additional functionality for mouse overs, presses, and releases as well as keyboard activity/events
Canvas	provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps
Checkbutton	set of boxes of which any number can be "checked" (similar to HTML checkbox input)
Entry	single-line text field with which to collect keyboard input (similar to HTML text input)
Frame	pure container for other widgets
Label	used to contain text or images
Listbox	presents user list of choices to pick from
Menu	actual list of choices "hanging" from a Menubutton that the user can choose from
Menubutton	provides infrastructure to contain menus (pulldown, cascading, etc.)
Message	similar to a Label, but displays multi-line text
Radiobutton	set of buttons of which only one can be "pressed" (similar to HTML radio input)
Scale	linear "slider" widget providing an exact value at current setting; with defined starting and ending values
Scrollbar	provides scrolling functionality to supporting widgets, i.e., Text, Canvas, Listbox, and Entry
Text	multi-line text field with which to collect (or display) text from user (similar to HTML textarea)
Toplevel	similar to a Frame, but provides a separate window container

Tkinter Examples**LabelWidget**

In [Example 18.1](#), we present tkhello1.py, the Tkinter version of "Hello World!" In particular, it shows you how a Tkinter application is set up and highlights the Labelwidget

Example 18.1. LabelWidget Demo (tkhello1.py)

Our first Tkinter example is... what else? "Hello World!" In particular, we introduce our first widget, the Label.

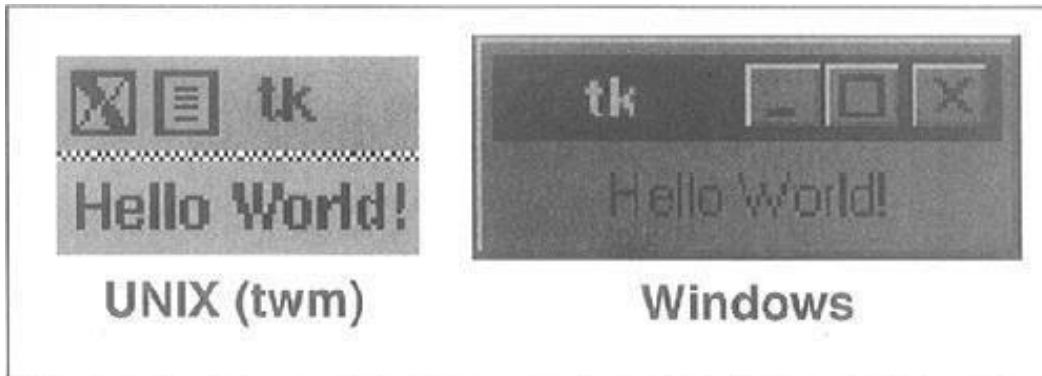
```

<$npage>
001 1      #!/usr/bin/env python
002 2
003 3      import Tkinter
004 4
005 5      top = Tkinter.Tk()
006 6      label = Tkinter.Label(top, text='Hello World!')
007 7      label.pack()
008 8      Tkinter.mainloop()
009 9      <$npage>

```

In the first line, we create our top-level window. That is followed by our Labelwidget containing the all-too-famous string. We instruct the packer to manage and display our widget, and finally call mainloop() to run our GUI application. [Figure 18-1](#) shows what you will see when you run this GUI application.

Figure 18-1. Tkinter LabelWidget (tkhello1.py)



ButtonWidget

The next example is pretty much the same as the first. However, instead of a simple text label, we will create a button instead. In [Example 18.2](#) is the source code for tkhello2.py:

Example 18.2. ButtonWidget Demo (tkhello2.py)

This example is exactly the same as tkhello1.py except that rather than using a Labelwidget, we create a Buttonwidget.

```
<$npage>
001 1      #!/usr/bin/env python
002 2
003 3      import Tkinter
004 4
005 5      top = Tkinter.Tk()
006 6      quit = Tkinter.Button(top, text='Hello World!',
007 7          command=top.quit)
008 8      quit.pack()
009 9      Tkinter.mainloop()
10     <$npage>
```

The first few lines are identical. Things differ only when we create the Button widget. Our button has one additional parameter, the Tkinter.quit() method. This installs a callback to our button so that if it is pressed (and released), the entire application will exit. The final two lines are the usual pack() and entering of the mainloop(). This simple button application is shown in [Figure 18-2](#).

Figure 18-2. Tkinter ButtonWidget (tkhello2.py)



Label and Button Widgets

We combine tkhello1.py and tkhello2.py into tkhello3.py, a script which has both a label and a button. In addition, we are providing more parameters now than before when we were comfortable using all the default arguments which are automatically set for us. The source for tkhello3.py is given in [Example 18.3](#).

Example 18.3. Label and Button Widget Demo (tkhello3.py)

This example features both a Label and a Button widget. Rather than primarily using default arguments when creating the widget, we are able to specify more now that we know more about Button widgets and how to configure them.

```
<$npage>
001 1      #!/usr/bin/env python
002 2
003 3      import Tkinter
004 4      top = Tkinter.Tk()
005 5
006 6      hello = Tkinter.Label(top, text='Hello World!') 007 7      hello.pack()
008 8
009 9      quit = Tkinter.Button(top, text='QUIT',
010 10         command=top.quit, bg='red', fg='white')
011 11      quit.pack(fill=Tkinter.X, expand=1)
012 12
013 13      Tkinter.mainloop()
014 14      <$npage>
```

Besides additional parameters for the widgets, we also see some arguments for the packer. The fill parameter tells the packer to let the QUIT button take up the rest of the horizontal real estate, and the expand parameter directs the packer to visually fill out the entire horizontal landscape, stretching the button to the left and right sides of the window.

As you can see in [Figure 18-3](#), without any other instructions to the packer, the widgets are placed vertically (on top of each other). Horizontal placement requires creating a new Frame object with which to add the buttons. That frame will take the place of the parent object as a single child object (see the buttons in the listdir.pymodule, [Example 18.5](#) in [Section 18.3.5](#)).

Figure 18-3. Tkinter Label and Button Widgets (tkhello3.py)



Label, Button, and Scale Widgets

Our final trivial example, `tkhello4.py`, involves the addition of a Scale widget. In particular, the Scale is used to interact with the Label widget. The Scale slider is a tool which controls the size of the text font in the Label widget. The greater the slider position, the larger the font, and the same goes for a lesser position, meaning a smaller font. The code for `tkhello4.py` is given in [Example 18.4](#).

New features of this script include a `resize()` callback function (lines 5–7), which is attached to the Scale. This is the code that is activated when the slider on the Scale is moved, resizing the size of the text in the Label.

We also define the size (250×150) of the top-level window (line 10). The final difference between this script and the first three is that we import the attributes from the Tkinter module into our namespace with **"from Tkinter import *."** This is mainly due to the fact that this application is larger and involves a large number of references to Tkinter attributes, which would otherwise require their fully-qualified names. The code is shortened a bit and perhaps may not wrap as many lines without importing all the attributes locally.

Example 18.4. Label, Button, and Scale Demo (`tkhello4.py`)

Our final introductory widget example introduces the Scale widget and highlights how widgets can "communicate" with each other using callbacks [such as `resize()`]. The text in the Label widget is affected by actions taken on the Scale widget.

```
<$nopage>
001 1      #!/usr/bin/env python
002 2
003 3      from Tkinter import * <$nopage> 004 4
05   5      def resize(ev=None):
06   6          label.config(font='Helvetica -%d bold' % \
07   007 7              scale.get())
008 8
009 9          top = Tk()
010 10 top.geometry('250×150')
011 11
12   12 label = Label(top, text='Hello World!',
13   13          font='Helvetica -12 bold')
14   14 label.pack(fill=Y, expand=1)
015 15
016 16 scale = Scale(top, from_=10, to=40, 017 17          orient=HORIZONTAL, command=resize)
18   18 scale.set(12)
19   19 scale.pack(fill=X, expand=1)
020 20
21   21 quit = Button(top, text='QUIT',
22   22          command=top.quit, activeforeground='white',
23   23          activebackground='red')
24   24 quit.pack()
025 25
26   26 mainloop()
27   <$nopage>
```

As you can see from [Figure 18-4](#), both the slider mechanism as well as the current set value show up in the main part of the window.

Figure 18-4. Tkinter Label, Button, and Scale Widgets (tkhello4.py)



Intermediate Tkinter Example

This application is a directory tree traversal tool. It starts in the current directory and provides a file listing. Double-clicking on any other directory in the list causes the tool to change to the new directory as well as replace the original file listing with the files from the new directory. The source code is given below as [Example 18.5](#).

Example 18.5. File System Traversal GUI (listdir.py)

This slightly more advanced GUI expands on the use of widgets, adding listboxes, text entry fields, and scrollbars to our repertoire. There are also a good number of callbacks such as mouse clicks, key presses, and scrollbar action.

```
<$nopage>
001 1  #!/usr/bin/env python
002 2
03  3  import os
04  4  from time import sleep
05  5  from Tkinter import * <$nopage>
06  006 6
007 7  class DirList:
008 8
09  9      def init (self, initdir=None):
10  10          self.top = Tk()
11  11          self.label = Label(self.top,
```

```

12      12      text='Directory Lister' + ' v1.1')
13      13      self.label.pack()
014 14
015 15      self.cwd=StringVar(self.top)
016 16
17      17      self.dirl = Label(self.top, fg='blue',
18      18          font=('Helvetica', 12, 'bold'))
19      19      self.dirl.pack()
020 20
21      21      self.dirfm = Frame(self.top)
22      22      self.dirsb = Scrollbar(self.dirfm)
23      23      self.dirsb.pack(side=RIGHT, fill=Y)
24      24      self.dirs = Listbox(self.dirfm, height=15,
25      25          width=50, yscrollcommand=self.dirsb.set)
26      26      self.dirs.bind('<Double-1>', self.setDirAndGo)
27      27      self.dirsb.config(command=self.dirs.yview)
28      28      self.dirs.pack(side=LEFT, fill=BOTH)
29      29      self.dirfm.pack()
030 30
31      31      self.dirn = Entry(self.top, width=50,
32      32          textvariable=self.cwd)
33      33      self.dirn.bind('<Return>', self.doLS)
34      34      self.dirn.pack()
035 35
36      36      self.bfm = Frame(self.top)
37      37      self.clr = Button(self.bfm, text='Clear',
38      38          command=self.clrDir,
39      39          activeforeground='white',
40      40          activebackground='blue')
41      41      self.ls = Button(self.bfm,
42      42          text='List.Directory',
43      43          command=self.doLS,
44      44          activeforeground='white',
45      45          activebackground='green')
46      46      self.quit = Button(self.bfm, text='Quit',
47      47          command=self.top.quit,
48      48          activeforeground='white',
49      49          activebackground='red')
50      50      self.clr.pack(side=LEFT)
51      51      self.ls.pack(side=LEFT)
52      52      self.quit.pack(side=LEFT)
53      53      self.bfm.pack()
054 54
055 55      if initdir:
056 56      self.cwd.set(os.curdir)
057 57      self.doLS()
058 58
059 59      def clrDir(self, ev=None):

```



```

060 60         self.cwd.set("")
061 61
062 62     def     setDirAndGo(self, ev=None):
063 63         self.last = self.cwd.get()
064 64         self.dirs.config(selectbackground='red')
065 65         check = self.dirs.get(self.dirs.curselection())
066 66         if
067         not check:
068 67         check = os.curdir
069 68         self.cwd.set(check)
070 69         self.doLS()
071 70
072 71     def     doLS(self, ev=None):
073 72         error = ""
074 73         tdir = self.cwd.get()
075 74         if
076         not tdir: tdir = os.curdir
077 75
078 76         if
079                                     not os.path.exists(tdir):
080 77         error = tdir + ': no such file'
081 78     elif
082                                     not os.path.isdir(tdir):
083 79         error = tdir + ': not a directory'
084 80
085 81         if     error:
086 82             self.cwd.set(error)
087 83             self.top.update()
088 84             sleep(2)
089 85             if not (hasattr(self, 'last') \
090 86             and self.last):
091 87                 self.last = os.curdir
092 88                 self.cwd.set(self.last)
093 89                 self.dirs.config(\
094 90                     selectbackground='LightSkyBlue')
095 91         self.top.update()
096 92     return <$nopage>
097 93
098 94         self.cwd.set(\
099 95         'FETCHING DIRECTORY CONTENTS...')
100 96         self.top.update()
101 97         dirlist = os.listdir(tdir)
102 98         dirlist.sort()
103 99         os.chdir(tdir)
104 100        self.dirl.config(text=os.getcwd())
105 101        self.dirs.delete(0, END)
106 102        self.dirs.insert(END, os.curdir)
107 103        self.dirs.insert(END, os.pardir)

```

```

108 104     for eachFile in dirlist:
109 105     self.dirs.insert(END, eachFile)
110 106     self.cwd.set(os.curdir)
111 107     self.dirs.config(\
112 108         selectbackground='LightSkyBlue')
113 109
114 110     def main():
115 111         d = DirList(os.curdir)
116         112     mainloop() 117 113
118 114     if name == '_main_':
119 115         main()
120     <$nopage>

```

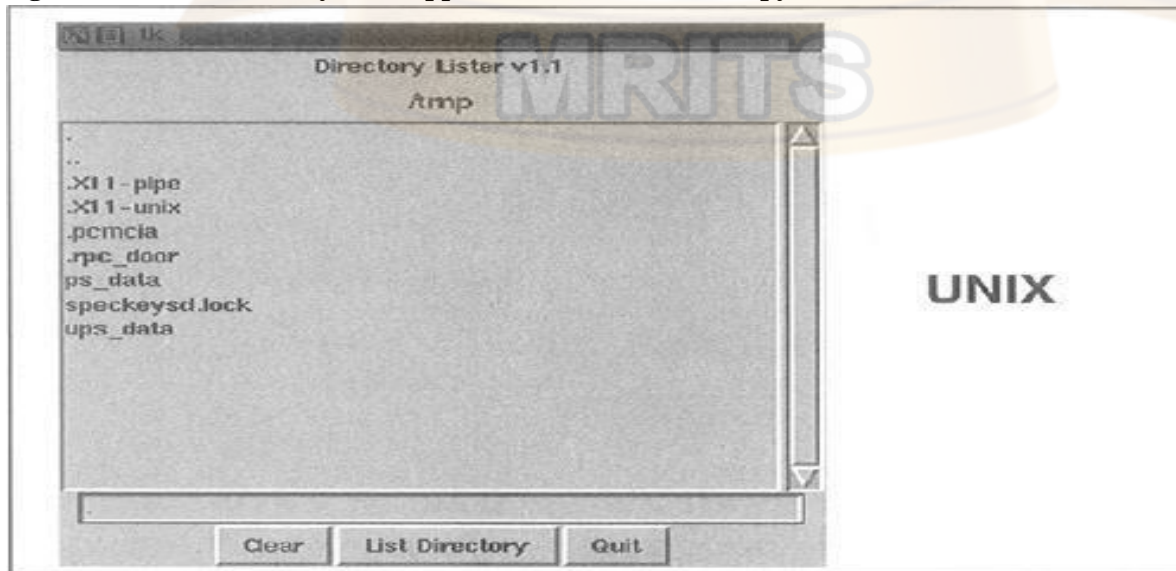
In [Figure18-5](#), we present what this GUI looks like in a Windows environment:

Figure 18-5. List Directory GUI Application in Windows (listdir.py)



The Unix version of this application is given in [Figure18-6](#).

Figure 18-6. List Directory GUI Application in UNIX (listdir.py)



Line-by-line explanation Lines 1–5

These first few lines contain the usual Unix start-up line and importation of the `osmodule`, the `time.sleep()` function, and all attributes of the `Tkintermodule`.

Lines 9–13

These lines define the constructor for the `DirListclass`, an object which represents our application. The first `Label` we create contains the main title of the application and the version number.

Lines 15–19

We declare a `Tk` variable named `cwd` to hold the name of the directory we are on—we will see where this comes in handy later. Another `Label` is created to display the name of the current directory.

Lines 21–30

This section defines the core part of our GUI, the `Listbox` `dirs`, which contain the list of files of the directory that is being listed. A `Scrollbar` is employed to allow the user to move through a listing if the number of files exceeds the size of the `Listbox`. Both of these widgets are contained in a `Frame` widget. `Listbox` entries have a callback (`setdirandgo`) tied to them using the `Listbox` `bind()` method.

Binding means to tie a keystroke, mouse action, or some other event to a call back to be executed when such an event is generated by the user. `setdirandgo()` will be called if any item in the `Listbox` is doubleclicked. The `Scrollbar` is tied to the `Listbox` by calling the `Scrollbar.config()` method.

Lines 32–35

We then create a text `Entry` field for the user to enter the name of the directory he or she wants to traverse to and see its files listed in the `Listbox`. We add a `RETURN` or `Enter` key binding to this text entry field so that the user can hit `RETURN` as an alternative to pressing a button. The same applies for the mouse binding we saw above in the `Listbox`. When the user doubleclicks on a `Listbox` item, it has the same effect as the user's entering the directory name manually into the text `Entry` field and pressing the "go" button.

Lines 37–54

We then define a `Buttonframe` (`bfm`) to hold our three buttons, a "clear" button (`clr`), "go" button (`ls`), and a "quit" button (`quit`). Each button has its own different configuration and callbacks, if pressed.

Lines 56–58

The final part of the constructor initializes the GUI program, starting with the current working directory.

Lines 60–61

The `clrDir()` method clears the `cwd` `Tk` string variable, which contains the current directory which is "active." This variable is used to keep track of what directory we are in and, more importantly, helps keep track of the previous directory in case errors arise. You will notice the `ev` variables in the callback functions with a default value of `None`. Any such values would be passed in by the windowing system. They may or may not be used in your callback.

Lines 63–71

The `setDirAndGo()` method sets the directory to traverse to and issues the call to the method that makes it all happen, `doLS()`.

Lines 73–108

`doLS()` is, by far, the key to this entire GUI application. It performs all the safety checks (e.g., is the destination a directory and does it exist?). If there is an error, the last directory is reset to be the current directory. If all goes well, it calls `os.listdir()` to get the actual set of files and replaces the listing in the `Listbox`. While the background work is going

on to pull in the new directory's information, the highlighted blue bar becomes a bright red. When the new directory has been installed, it reverts to blue.

Lines 110–115

The last pieces of code in `listdir.py` represent the main part of the code. `main()` is executed only if this script is invoked directly, and when `main()` runs, it creates the GUI application, then calls `mainloop()` to start the GUI, which is passed control of the application.

We leave all other aspects of the application as an exercise to the reader, recommending that it is easier to view the entire application as a combination of a set of widgets and functionality. If you see the individual pieces clearly, then the entire script will not appear as daunting.

We hope that we have given you a good introduction to GUI programming with Python and Tkinter. Remember that the best way to get familiar with Tkinter programming is by practicing and stealing a few examples! The Python distribution comes with a large number of demonstration applications (see the Demo directory) that you can study. And as we mentioned earlier, there is also an entire text devoted to Tkinter programming.

One final note: do you still doubt the ability of Tkinter to produce a commercial application? Take a close look at IDLE. IDLE itself is a Tkinter application (written by Guido)!

Related Modules and Other GUIs

There are other GUI development systems which can be used with Python. We present the appropriate modules along with their corresponding window systems in [Table 18.2](#).

Table 18.2. GUI Systems Available Under Python

<i>GUI Module or System</i>	<i>Description</i>
Other Tkinter Modules	
<code>Pmw</code>	Python Mega Widgets
Open Source	
<code>wxPython</code>	<code>wxWindows</code>
<code>PyGTK</code>	GTK+/GNOME/Glade/GIMP
<code>PyQt/PyKDE</code>	Qt/KDE
Commercial	
<code>win32ui</code>	Microsoft MFC
<code>swing</code>	Sun Microsystems Java/Swing

Explain about Web Programming with python.

Web Programming

Introduction

No Python book would be complete without discussing how to do Web programming, one of the main avenues in which people discover Python. In fact, one of the very first Python books was named, "Internet Programming with Python" (unfortunately out-of-print). This introductory chapter on web programming will give you a quick and high-level overview of the kinds of things you can do with Python on the Internet, from Web surfing to creating user feedback forms, from recognizing Uniform Resource Locators to generating dynamic Web page output.

Web Surfing: Client-Server Computing (Again?!?)

Web surfing falls under the same client-server architecture umbrella that we have seen repeatedly. This time, *Web clients are browsers*; applications allow users to seek documents on the World Wide Web. On the other side are *Web servers*, processes which run on an information provider's host computers. These servers wait for clients and their document requests, process them, and return the requested data. As with most servers in a client-server system, Web servers are designed to run "forever." The Web surfing experience is best illustrated by [Figure 19-1](#). Here, a user runs a Web client program such as a browser and makes a connection to a Web server elsewhere on the Internet to obtain their information.

Figure 19-1. Web Client and Web Server on the Internet. A client sends a request out over the Internet to the server, which then responds with the requested data back to the client.



Clients may issue a variety of requests to Web servers. Such requests may include obtaining a Web page for viewing or submitting a form with data for processing. The request is then serviced by the Web server, and the reply comes back to the client in a special format for display purposes.

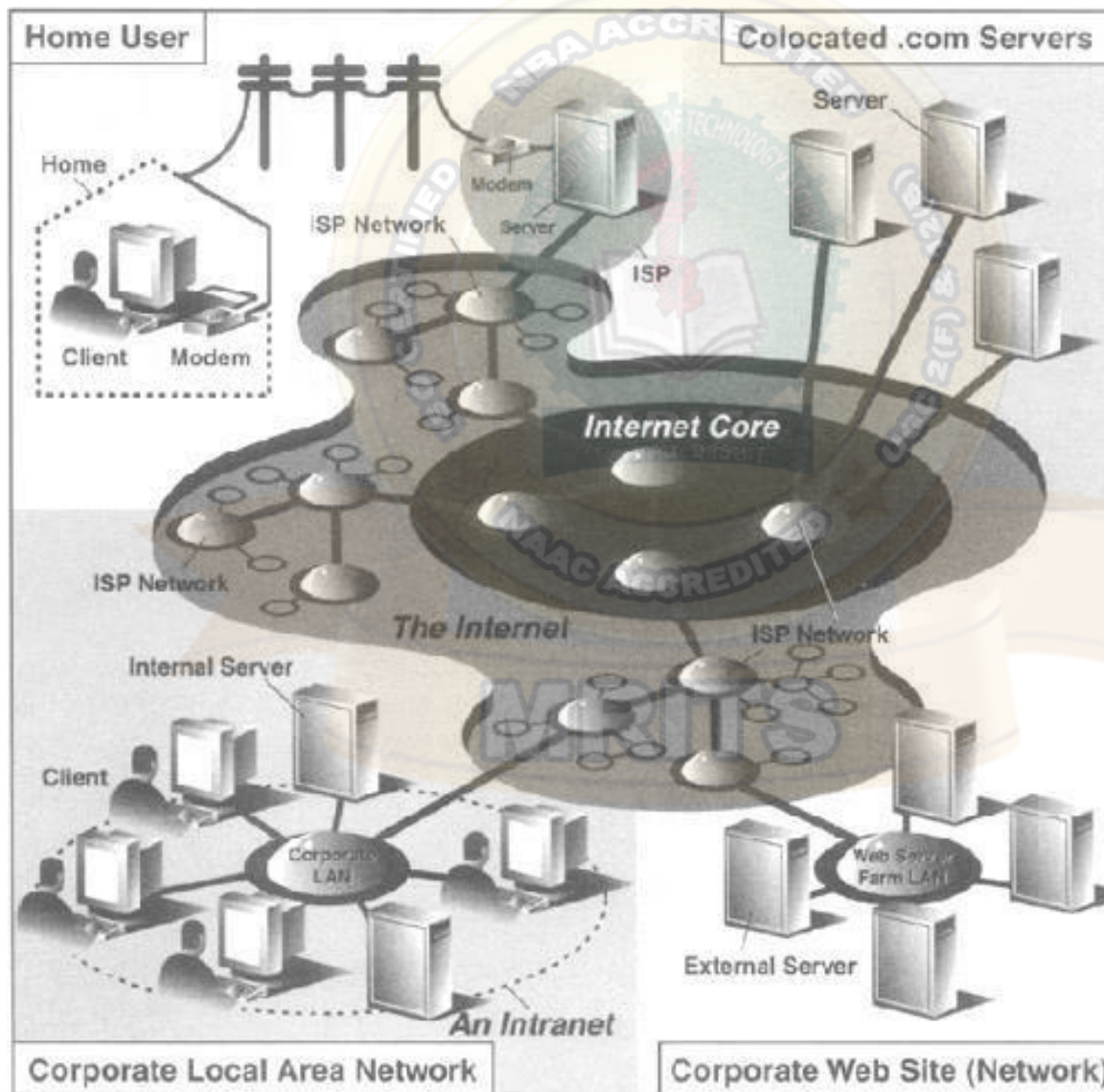
The "language" which is spoken by Web clients and servers, the standard protocol used for Web communication, is called *HTTP*, which stands for *HyperText Transfer Protocol*. HTTP is written "on top of" the TCP and IP protocol suite, meaning that it relies on TCP and IP to carry out its lower-level communication functionality. Its responsibility is not to route or deliver messages—TCP and IP handle that—but to respond to client requests.

HTTP is known as a "stateless" protocol because it does not keep track of information from one client request to the next, similar to the client-server architecture we have seen so far. The server stays running, but client interactions are singular events structured in such a way that once a client request is serviced, it quits. New requests can always be sent, but they are considered separate service requests. Because of the lack of context per request, you may notice that some URLs have a long set of variables and values chained as part of the request to provide some sort of state information. Another alternative is the use of "cookies"—static data stored on the client side which generally contains state information as well. In later parts of this chapter, we will look at how to use both long URLs and cookies to maintain state information.

The Internet

The Internet is a moving and fluctuating "cloud" or "pond" of interconnected clients and servers scattered around the globe. Communication between client and server consists of a series of connections from one lily pad on the pond to another, with the last step connecting to the server. As a client user, all this detail is kept hidden from your view. The abstraction is to have a direct connection between you the client and the server you are "visiting," but the underlying HTTP, TCP, and IP protocols are hidden underneath, doing all of the dirty work. Information regarding the intermediate "nodes" is of no concern or consequence to the general user anyway, so it's good that the implementation is hidden. [Figure 19-2](#) shows an expanded view of the Internet.

Figure 19-2. A Grand View of the Internet. The left side illustrates where you would find Web clients while the right side hints as to where Web servers are typically located.



As you can see from the figure, the Internet is made up of multiply-interconnected networks, all working with some sense of (perhaps disjointed) harmony. The left half of the diagram is focused on the Web clients, users who are either at home dialed-in to their *Internet Service Provider* (ISP) or at work on their company's *Local Area Network* (LAN). The right hand side of the diagram concentrates more on Web servers and where they can be found. Corporations with

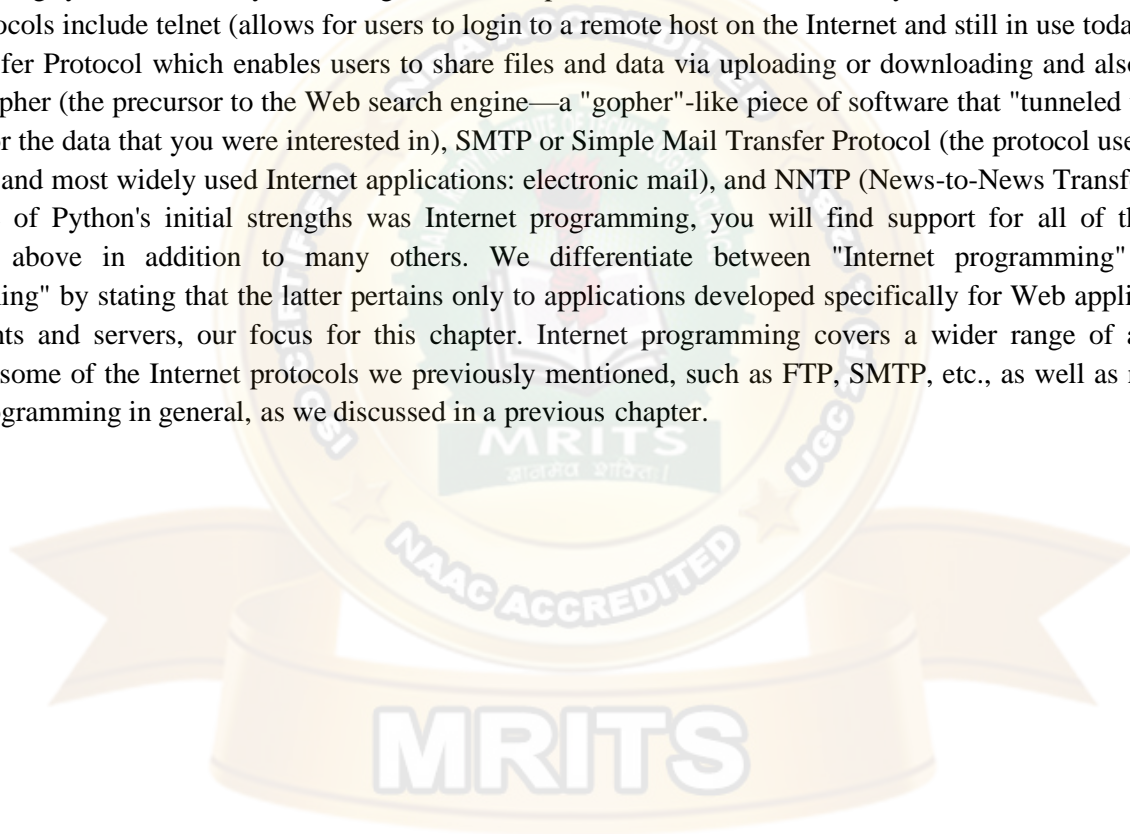
larger Web sites will typically have an entire "Web server farm" located at their ISP. Such physical placement is called *colocation*, meaning that a company's servers are "co-located" at an ISP along with machines from other corporate customers.

These servers are either all providing different data to clients or are part of a redundant system with duplicated information designed for heavy demand (high number of clients). Smaller corporate Web sites may not require as much hardware and networking gear, and hence, may only have one or several colocated servers at their ISP.

In either case, most colocated servers are stored with a larger ISP sitting on a network *backbone*, meaning that they have a "fatter" (meaning wider) and presumably faster connection to the Internet—closer to the "core" of the Internet, if you will. This permits clients to access the servers quickly—being on a backbone means clients do not have to hop across as many networks to access a server, thus allowing more clients to be serviced within a given time period.

One should also keep in mind that although Web surfing is the most common Internet application, it is not the only one and is certainly not the oldest. The Internet predates the Web by almost three decades. Before the Web, the Internet was mainly used for educational and research purposes. Most of the systems on the Internet run Unix, a multi-user operating system, and many of the original Internet protocols are still around today.

Such protocols include telnet (allows for users to login to a remote host on the Internet and still in use today), FTP (the File Transfer Protocol which enables users to share files and data via uploading or downloading and also still in use today), gopher (the precursor to the Web search engine—a "gopher"-like piece of software that "tunneled the Internet" looking for the data that you were interested in), SMTP or Simple Mail Transfer Protocol (the protocol used for one of the oldest and most widely used Internet applications: electronic mail), and NNTP (News-to-News Transfer Protocol). Since one of Python's initial strengths was Internet programming, you will find support for all of the protocols discussed above in addition to many others. We differentiate between "Internet programming" and "Web programming" by stating that the latter pertains only to applications developed specifically for Web applications, i.e., Web clients and servers, our focus for this chapter. Internet programming covers a wider range of applications, including some of the Internet protocols we previously mentioned, such as FTP, SMTP, etc., as well as network and socket programming in general, as we discussed in a previous chapter.



Web Surfing with Python: Creating Simple Web Clients

One thing to keep in mind is that a browser is only one type of Web client. Any application that makes a request for data from a Web server is considered a "client." Yes, it is possible to create other clients which retrieve documents or data off the Internet. One important reason to do this is that a browser provides only limited capacity, i.e., it is used primarily for viewing and interacting with Web sites. A client program, on the other hand, has the ability to do more—it can not only download data, it can also store it, manipulate it, or perhaps even transmit it to another location or application.

Applications which use the `urllib` module to download or access information on the Web [using either `urllib.urlopen()` or `urllib.urlretrieve()`] can be considered a simple Web client. All you need to do is provide a valid Web address.

Uniform Resource Locators

Simple Web surfing involves using Web addresses called *Uniform Resource Locators* (URLs). Such addresses are used to locate a document on the Web or to call a CGI program to generate a document for your client. URLs are part of a larger set of identifiers known as *URIs* (*Uniform Resource Identifiers*). This superset was created in anticipation of other naming conventions which have yet to be developed. A URL is simply a URI which uses an existing protocol or scheme (i.e., `http`, `ftp`, etc.) as part of its addressing. To complete this picture, we'll add that non-URL URIs are sometimes known as *URNs* (*Uniform Resource Names*), but because URLs are the only URIs in use today, you really don't hear much about URIs or URNs.

Like street addresses, Web addresses have some structure. An American street address usually is of the form "number street designation," i.e., 123 Main Street. It differs from other countries, which have their own rules. A URL is of the format:

`prot_sch://net_loc/path;params?query#frag`

[Table 19.1](#) describes each of the components.

Table 19.1. Web Address Components

<i>URL Component</i>	<i>Description</i>
<code>prot_sch</code>	network protocol or download scheme
<code>net_loc</code>	location of server (and perhaps user information)
<code>path</code>	slash (/) delimited path to file or CGI application
<code>params</code>	optional parameters
<code>query</code>	ampersand (&) delimited set of "key=value" pairs
<code>frag</code>	fragment to a specific anchor within document

`net_loc` can be broken down into several more components, some required, others optional. The `net_loc` string looks like this:

`user:passwd@host:port`

These individual components are described in [Table 19.2](#).

Table 19.2. Network Location Components

<i>net_loc Component</i>	<i>Description</i>
<code>user</code>	user name or login
<code>passwd</code>	user password
<code>host</code>	name or address of machine running Web server [required]
<code>port</code>	port number (if not 80, the default)

Of the four, the host name is the most important. The port number is necessary only if the Web server is running on a different port number from the default. (If you aren't sure what a port number is, go back to [Chapter 16](#).)

User names and perhaps passwords are used only when making FTP connections, and even then, they usually aren't necessary because the majority of such connections are "anonymous."

Python supplies two different modules, each dealing with URLs in completely different functionality and capacities. One is `urlparse`, and the other is `urllib`. We will briefly introduce some of their functions here.

urlparseModule

The urlparse module provides basic functionality with which to manipulate URL strings. These functions include urlparse(),urlunparse(),and urljoin().

urlparse.urlparse()

urlparse() breaks up a URL string into some of the major components described above and has the following syntax:

```
urlparse(urlstr, defProtSch=None, allowFrag=None)
```

urlparse() parses *urlstr* into a 6-tuple (prot_sch, net_loc, path, params, query, frag). Each of these components has been described above. *defProtSch* indicates a default network protocol or download scheme in case one is not provided in *urlstr*. *allowFrag* is a flag that signals whether or not a fragment part of a URL is allowed. Here is what urlparse() outputs when given a URL:

```
>>> urlparse.urlparse('http://www.python.org/doc/FAQ.html') ('http', 'www.python.org', '/doc/FAQ.html', '', '', '')
```

urlparse.urlunparse()

urlunparse() does the exact opposite of urlparse()—it merges a 6-tuple (prot_sch, net_loc, path, params, query, frag)—*urltup*, which could be the output of urlparse(), into a single URL string and returns it. Accordingly, we state the following equivalence:

```
urlunparse(urlparse(urlstr)) = urlstr
```

You may have already surmised that the syntax of urlunparse() is as follows:

```
urlunparse(urltup)
```

urlparse.urljoin()

The urljoin() function is useful in cases where many related URLs are needed, for example, the URLs for a set of pages to be generated for a Web site. The syntax for urljoin() is:

```
urljoin(baseurl, newurl, allowFrag=None)
```

urljoin() takes *baseurl* and joins its base path (*net_loc* plus the full path up to, but not including, a file at the end) with *newurl*. For example:

```
>>> urlparse.urljoin('http://www.python.org/doc/FAQ.html', \
... 'current/lib/lib.htm') 'http://www.python.org/doc/current/lib/lib.html'
```

A summary of the functions in urlparse can be found in [Table 19.3](#)

Table 19.3. Core urlparseModule Functions

urlparse Functions	Description
<code>urlparse(urlstr, defProtSch=None, allowFrag=None)</code>	parses <i>urlstr</i> into separate components, using <i>defProtSch</i> if the protocol or scheme is not given in <i>urlstr</i> ; <i>allowFrag</i> determines whether a URL fragment is allowed
<code>urlunparse(urltup)</code>	unparses a tuple of URL data (<i>urltup</i>) into a single URL string
<code>urljoin(baseurl, newurl, allowFrag=None)</code>	merges the base part of the <i>baseurl</i> URL with <i>newurl</i> to form a complete URL; <i>allowFrag</i> is the same as for <code>urlparse()</code>

urllibModule

The urllib module provides functions to download data from given URLs as well as encoding and decoding strings to make them suitable for including as part of valid URL strings. The functions which we will be looking at in this upcoming section include: urlopen(), urlretrieve(), quote(), quote_plus(), unquote(), unquote_plus(), and urlencode().

We will also look at some of the methods available to the file-like object returned by urlopen().

urllib.urlopen()

urlopen() opens a Web connection to the given URL string and returns a file-like object. It has the following syntax:

```
urlopen(urlstr, postData=None)
```

urlopen() opens the URL pointed to by *urlstr*. If no protocol or download scheme is given, or if a "file" scheme is passed in, urlopen() will open a local file.

For all HTTP requests, the normal request type is "GET." In these cases, the query string provided to the Web server (key-value pairs encoded or "quoted," such as the string output of the urlencode() function [see below]), should be given as part of *urlstr*.

If the "POST" request method is desired, then the query string (again encoded) should be placed in the *postData* variable. (For more information regarding the GET and POST request methods, refer to any general documentation or texts on programming CGI applications—which we will also discuss below. GET and POST requests are the two ways to "upload" data to a Web server.

When a successful connection is made, urlopen() returns a file-like object as if the destination was a file opened in read mode. If our file object is *f*, for example, then our "handle" would support the expected read methods such as *f.read()*, *f.readline()*, *f.readlines()*, *f.close()*, and *f.fileno()*.

In addition, a *f.info()* method is available which returns the *MIME (Multipurpose Internet Mail Extension)* headers. Such headers give the browser information regarding which application can view returned file types. For example, the browser itself can view HTML (*Hypertext Markup Language*) or plain text type files as well as *GIF (Graphics Interchange Format)* and *JPEG (Joint Photographic Experts Group)* graphics files. Other files such as multimedia or specific document types require external applications in order to view.

Finally, a *geturl()* method exists to obtain the true URL of the final opened destination, taking into consideration any redirection which may have occurred. A summary of these file-like object methods is given in [Table 19.4](#).

Table 19.4. urllib.urlopen() File-like Object Methods

urlopen () Object Methods	Description
<i>f.read ([bytes])</i>	reads all or bytes <i>bytes</i> from <i>f</i>
<i>f.readline ()</i>	reads a single line from <i>f</i>
<i>f.readlines ()</i>	reads a all lines from <i>f</i> into a list
<i>f.close ()</i>	closes URL connection for <i>f</i>
<i>f.fileno ()</i>	returns file number of <i>f</i>
<i>f.info ()</i>	gets MIME headers of <i>f</i>
<i>f.geturl ()</i>	returns true URL opened for <i>f</i>

urllib.urlretrieve()

urlretrieve() will do some quick and dirty work for you if you are interested in working with a URL document as a whole. Here is the syntax for urlretrieve():

```
urlretrieve(urlstr, localfile=None, downloadStatusHook=None)
```

Rather than reading from the URL like urlopen() does, urlretrieve() will simply download the entire HTML file located at *urlstr* to your local disk. It will store the downloaded data into *localfile* if given or a temporary file if not. If the file has already been copied from the Internet or if the file is local, no subsequent downloading will occur.

The *downloadStatusHook*, if provided, is a function that is called after each block of data has been downloaded and delivered. It is called with the following three arguments: number of blocks read so far, the block size in bytes, and the total (byte) size of the file. This is very useful if you are implementing "download status" information to the user in a text-based or graphical display.

urlretrieve() returns a 2-tuple, (*filename*, *mime_hdrs*). *filename* is the name of the local file containing the downloaded data. *mime_hdrs* is the set of MIME headers returned by the responding Web server. For more information, see the Message class of the mimetools module. *mime_hdrs* is None for local files.

For an example using urlretrieve(), take a look at [Example 11.2](#) (grabweb.py).

urllib.quote()and urllib.quote_plus()

The quote*() functions take URL data and "encodes" them so that they are "fit" for inclusion as part of a URL string. In particular, certain special characters that are unprintable or cannot be part of valid URLs acceptable to a Web server must be converted. This is what the quote*() functions do for you. Both quote*() functions have the following syntax:

```
quote(urldata, safe='/')
```

Characters that are never converted include commas, underscores, periods and dashes as well as alphanumerics. All others are subject to conversion. In particular, the disallowed characters are changed to their hexadecimal ordinal equivalents prepended with a percent sign (%), i.e., "%xx" where "xx" is the hexadecimal representation of a character's ASCII value. When calling quote*(), the urldata string is converted to an equivalent string that can be part of a URL string. The safe string should contain a set of characters which should also not be converted.

The default is the slash (/). quote_plus() is similar to quote() except that it also encodes spaces to plus signs (+). Here is an example using quote()vs. quote_plus():

```
>>> name = 'joe mama'
>>> number = 6
>>> base = 'http://www/~foo/cgi-bin/s.py'
>>> final = '%s?name=%s&num=%d' % (base, name, number)
>>> final
'http://www/~foo/cgi-bin/s.py?name=joe mama&num=6'
>>>
>>> urllib.quote(final)
'http:%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe%20mama%26num%3d6'
>>>
>>> urllib.quote_plus(final)
'http%3a//www/%7efoo/cgi-bin/s.py%3fname%3djoe+mama%26num%3d6'
```

urllib.unquote()and urllib.unquote_plus()

As you have probably guessed, the unquote*() functions do the exact opposite of the quote*() functions—they convert all characters encoded in the "%xx" fashion to their ASCII equivalents. The syntax of unquote*()is as follows:

```
unquote*(urldata)
```

Calling unquote() will decode all URL-encoded characters in urldata and return the resulting string. unquote_plus() will also convert plus signs back to space characters.

urllib.urlencode()

urlencode(), recently added to Python (as of version 1.5.2) takes a dictionary of key-value pairs and encodes them to be included as part of a query in a CGI request URL string. The pairs are in "key=value" format and are delimited by ampersands (&). Furthermore, the keys and their values are sent to quote_plus() for proper encoding. Here is an example output from urlencode():

```
>>> aDict = { 'name': 'Georgina Garcia', 'hmdir': '~ggarcia' }
>>> urllib.urlencode(aDict) 'name=Georgina+Garcia&hmdir=%7eggarcia'
```

There are other functions in urllib and urlparse which we did not have the opportunity to cover here. Refer to the documentation for more information.

Secure Socket Layer support

The urllib module has been modified for Python 1.6 so that it now supports opening HTTP connections using the Secure Socket Layer (SSL). The core change to add SSL is implemented in the socket module. Consequently, the urllib and httplib modules were updated to support URLs using the "https" connection scheme. Note, however, that as of time of publication, only HTTP requests using SSL have been implemented. The future may see additional updates to the other protocols supported by the urllib module, such as FTP.

A summary of the urllib functions discussed in this section can be found in [Table 19.5](#).

Table 19.5. Core urllibModule Functions

urllib Functions	Description
<code>urlopen(urlstr, postData=None)</code>	opens the URL <i>urlstr</i> , sending the query data in <i>postData</i> if a POST request
<code>urlretrieve(urlstr, localfile=None, downloadStatusHook=None)</code>	downloads the file located at the <i>urlstr</i> URL to <i>localfile</i> or a temporary file if <i>localfile</i> not given; if present, <i>downloadStatusHook</i> is a function which can receive download statistics
<code>quote(urldata, safe='/')</code>	encodes invalid URL characters of <i>urldata</i> ; characters in <i>safe</i> string are also not encoded
<code>quote_plus(urldata, safe='/')</code>	same as <code>quote()</code> except encodes spaces as plus signs
<code>unquote(urldata)</code>	decodes encoded characters of <i>urldata</i>
<code>unquote_plus(urldata)</code>	same as <code>unquote()</code> but converts plus signs to spaces
<code>urlencode(dict)</code>	encodes the key-value pairs of <i>dict</i> into a valid string for CGI queries and encodes the key and value strings with <code>quote_plus()</code>

Write a note on Advanced Web Clients with python.

Advanced Web Clients

Web browsers are basic Web clients. They are used primarily for searching and downloading documents from the Web. Advanced clients of the Web are those applications which do more than download single documents from the Internet.

One example of an advanced Web client is a *crawler* (a.k.a. *spider*, *robot*). These are programs which explore and download pages from the Internet for different reasons, some of which include:

- Indexing or cataloging into a large search engine such as Google, Alta Vista, or Yahoo!,
- Offline browsing—downloading documents onto a local hard disk and rearranging hyperlinks to create almost a mirror image for local browsing,
- Downloading and storing for historical or archival purposes, or
- Web page caching to save superfluous downloading time on Web site revisits.

The crawler we present below, `crawl.py`, takes a starting Web address (URL), downloads that page and all other pages whose links appear in succeeding pages, but only those which are in the same domain as the starting page. Without such limitations, you will run out of disk space! The source for `crawl.py` follows:

Example 19.1. An Advanced Web Client: a Web Crawler (`crawl.py`)

The crawler consists of two classes, one to manage the entire crawling process (Crawler), and one to retrieve and parse each downloaded Web page (Retriever).

```

<$nopage>
001 1  #!/usr/bin/env python
002 2
03   3  from sys import argv
04   4  from os import makedirs, unlink
05   5      from os.path import dirname, exists, isdir, splitext
06   006 6  from string import replace, find, lower
07   7  from htmllib import HTMLParser
08   8  from urllib import urlretrieve
09   9  from urlparse import urlparse, urljoin
10   10   from formatter import DumbWriter, AbstractFormatter

```



```

11 011 11 from cStringIO import StringIO
12 12
13 13 class Retriever:           # download Web pages
14 14
15 15     def init (self, url):
16 16         self.url = url
17 17         self.file = self.filename(url)
18 18
19 19     def filename(self, url, deffile='index.htm'):
20 20         parsedurl = urlparse(url, 'http:', 0)# parse path
21 21         path = parsedurl[1] + parsedurl[2]
22 22         text = splitext(path)
23 23         if ext[1] == "":           # no file, use default
24 24         if newpath[-1] == '/':
25 25             path = path + deffile
26 26         else: <$nopcode>
27 27             path = path + '/' + deffile
28 28         dir = dirname(path)
29 29         if not isdir(dir):           # create archive dir if nec.
30 30         if exists(dir): unlink(dir)
31 31         makedirs(dir)
32 32         return path
33 33
34 34     def download(self):           # download Web page
35 35         try: <$nopcode>
36 36             retval = urlretrieve(self.url, self.file)
37 37         except IOError:
38 38             retval = ("*** ERROR:           invalid URL "%s" %\
39 39             self.url,)
40 40         return retval
41 41
42 42     def parseAndGetLinks(self):   # parse HTML, save links
43 43         self.parser = HTMLParser(AbstractFormatter(\
44 44         DumbWriter(StringIO()))
45 45         self.parser.feed(open(self.file).read())
46 46         self.parser.close()
47 47         return self.parser.anchorlist
48 48
49 49 class Crawler:                 # manage entire crawling process
50 50
51 51     count = 0                   # static downloaded page counter
52 52
53 53     def init (self, url):
54 54         self.q = [url]
55 55         self.seen = []
56 56         self.dom = urlparse(url)[1]
57 57
58 58     def getPage(self, url):

```

```

059 59     r = Retriever(url)
060 60     retval = r.download()
061 61     if retval[0] == '*':                               # error situation, do not parse
062 62         print retval, '... skipping parse'
063 63         return <$npage>
064 64     Crawler.count = Crawler.count + 1
065 65     print '\n(', Crawler.count, ')'
066 66     print 'URL:', url
067 67     print 'FILE:', retval[0]
068 68     self.seen.append(url)
069 69
070 70         links = r.parseAndGetLinks() # get and process links
071 71         for         eachLink in links:
072 72             if eachLink[:4] != 'http' and \
073 73             find(eachLink, '/') == -1:
074 74                 eachLink = urljoin(url, eachLink)
075 75         print '* ', eachLink,
076 76
077 77         if find(lower(eachLink), 'mailto:') != -1:
078 78         print '... discarded, mailto link'
079 79         continue <$npage>
080 80
081 81         if eachLink not in self.seen:
082 82         if find(eachLink, self.dom) == -1:
083 83             print '... discarded, not in domain'
084 84         else: <$npage>
085 85         if eachLink not in self.q:
086 86         self.q.append(eachLink)
087 87         print '... new, added to Q'
088 88         else: <$npage>
089 89             print '... discarded, already in Q'
090 90         else: <$npage>
091 91             print '... discarded, already processed'
092 92
093 93     def go(self):# process links in queue
094 94         while self.q:
095 95 url = self.q.pop()
096 96 self.getPage(url)
097 97
098 98 def main():
099 99 if len(argv) > 1:
100 100         url = argv[1]
101 101         else: <$npage>
102 102         try: <$npage>
103 103             url = raw_input('Enter starting URL: ')
104 104         except (KeyboardInterrupt, EOFError):
105 105             url = ""
106 106
107 107         if not url: return <$npage>

```

```

108      108  robot = Crawler(url)
109      109  robot.go()
110      110 110
111 111 if name_          == '_main_':
112 112     main()
113 <$nopage>

```

Line-by-line (Class-by-class) explanation:

Lines 1– 11

The top part of the script consists of the standard Python Unix start-up line and the importation of various module attributes which are employed in this application.

Lines 13 – 47

The Retriever class has the responsibility of downloading pages from the Web and parsing the links located within each document, adding them to the "to-do" queue if necessary. A Retriever instance object is created for each page which is downloaded from the net. Retriever consists of several methods to aid in its functionality: a constructor (`_init_()`), `filename()`, `download()`, and `parseAndGetLinks()`.

The `filename()` method takes the given URL and comes up with a safe and sane corresponding file name to store locally. Basically, it removes the "http://" prefix from the URL and uses the remaining part as the file name, creating any directory paths necessary. URLs without trailing file names will be given a default file name of "index.htm." (This name can be overridden in the call to `filename()`).

The constructor instantiates a Retriever object and stores both the URL string and the corresponding file name returned by `filename()` as local attributes.

The `download()` method, as you may imagine, actually goes out to the net to download the page with the given link. It calls `urllib.urlretrieve()` with the URL and saves it to the filename (the one returned by `filename()`). If the download was successful, the `parse()` method is called to parse the page just copied from the network, otherwise an error string is returned. If the Crawler determines that no error has occurred, it will invoke the `parseAndGetLinks()` method to parse newly-downloaded page and determine the cause of action for each link located on that page.

Lines 49 – 96

The Crawler class is the "star" of the show, managing the entire crawling process, thus only one instance is created for each invocation of our script. The Crawler consists of three items stored by the constructor during the instantiation phase, the first of which is `q`, a queue of links to download. Such a list will fluctuate during execution, shrinking as each page is processed and grown as new links are discovered within each downloaded page.

The other two data values for the Crawler include `seen`, a list of all the links which "we have seen" (downloaded) already. And finally, we store the domain name for the main link, `dom`, and use that value to determine whether any succeeding links are part of the same domain.

Crawler also has of a static data item named `count`. The purpose of this counter is just to keep track of the number of objects we have downloaded from the net. It is incremented for every page successfully download.

Crawler has a pair of other methods in addition to its constructor, `getPage()` and `go()`. `go()` is simply the method that is used to start the Crawler and is called from the main body of code. `go()` consists of a loop that will continue to execute as long as there are new links in the queue which need to be downloaded. The workhorse of this class though, is the `getPage()` method. `getPage()` instantiates a Retriever object with the first link and lets it go off to the races. If the page was downloaded successfully, the counter is incremented and the link added to the "already seen" list. It looks recursively at all the links featured inside each downloaded page and determine whether any more links should be added to the queue. The main loop in `go()` will continue to process links until the queue is empty, at which time victory is declared.

Links which are: part of another domain, have already been downloaded, are already in the queue waiting to be

processed, or are "mailto:" links are ignored and not added to the queue.

Lines 98 – 112

main() is executed if this script is invoked directly and is the starting point of execution. Other modules which import crawl.py will need to invoke main() to begin processing. main() needs a URL to begin processing, If one is given on the command-line (for example which this script is invoked directly), it will just go with the one given. Otherwise, the script enters interactive mode prompting the user for a starting URL. With a starting link in hand, the Crawler is instantiated and away we go.

One sample invocation of crawl.pymay look like:

```
% crawl.py
```

```
Enter starting URL: http://www.null.com/home/index.html
```

```
( 1 )
```

```
URL: http://www.null.com/home/index.html FILE: www.null.com/home/index.html
```

- * http://www.null.com/home/overview.html ... new, added to Q
- * http://www.null.com/home/synopsis.html ... new, added to Q
- * http://www.null.com/home/order.html ... new, added to Q
- * mailto:postmaster@null.com ... discarded, mailto link
- * http://www.null.com/home/overview.html ... discarded, already in Q
- * http://www.null.com/home/synopsis.html ... discarded, already in Q
- * http://www.null.com/home/order.html ... discarded, already in Q
- * mailto:postmaster@null.com ... discarded, mailto link
- * http://bogus.com/index.html ... discarded, not in domain

```
( 2 )
```

```
URL: http://www.null.com/home/order.html FILE: www.null.com/home/order.html
```

- * mailto:postmaster@null.com ... discarded, mailto link
- * http://www.null.com/home/index.html ... discarded, already processed
- * http://www.null.com/home/synopsis.html ... discarded, already in Q
- * http://www.null.com/home/overview.html ... discarded, already in Q

```
( 3 )
```

```
URL: http://www.null.com/home/synopsis.html FILE: www.null.com/home/synopsis.html
```

- * http://www.null.com/home/index.html ... discarded, already processed
- * http://www.null.com/home/order.html ... discarded, already processed
- * http://www.null.com/home/overview.html ... discarded, already in Q

```
( 4 )
```

```
URL: http://www.null.com/home/overview.html FILE: www.null.com/home/overview.html
```

- * http://www.null.com/home/synopsis.html ... discarded, already processed
- * http://www.null.com/home/index.html ... discarded, already processed
- * http://www.null.com/home/synopsis.html ... discarded, already processed
- * http://www.null.com/home/order.html ... discarded, already processed

After execution, a <http://www.null.com> directory would be created in the local file system, with a home subdirectory. Within home,all the HTML files processed will be found there.

Explain in detail about CGI.

CGI: Helping Web Servers Process Client Data

Introduction to CGI

The Web was initially developed to be a global online repository or archive of (mostly educational and research-oriented) documents. Such pieces of information generally come in the form of static text and usually in HTML (*HyperText Markup Language*). [Many documents also exist in plain text, Adobe *Portable Document Format* (PDF), or *Extensible Markup Language* (XML) format, a generalized markup language.]

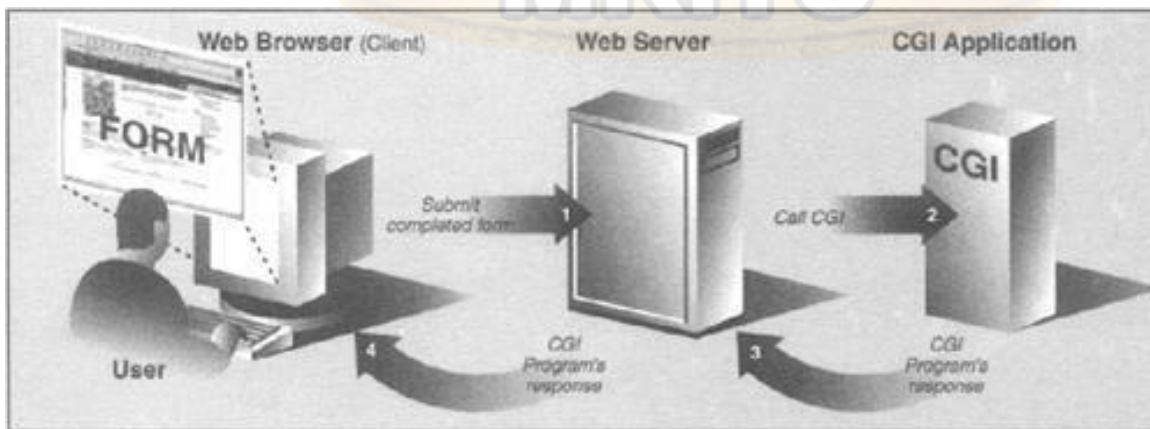
HTML is not as much of a *language* as it is a text formatter, indicating changes in font types, sizes, and styles. The main feature of HTML is in its hypertext capability, document text that is in one way or another highlighted to point to another document in a related context to the original. Such a document can be accessed by a mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when and if requested.

As the Internet and Web services evolved, there grew a need to process user input. Online retailers needed to be able to take individual orders, and online banks and search engine portals needed to create accounts for individual users. Thus fill-out forms were invented, and became the only way a Web site can get specific information from users (until Java applets came along). This, in turn, required the HTML now be generated on the fly, for each client submitting user-specific data.

Now Web servers are only really good at one thing, getting a user request for a file and returning that file (i.e., an HTML file) to the client. They do not have the "brains" to be able to deal with user-specific data such as those which come from fields. Not being their responsibility, Web servers farm out such requests to external applications which create the dynamically-generated HTML that is returned to the client.

The entire process begins when the Web server receives a client request (i.e., GET or POST) and calls the appropriate application. It then waits for the resulting HTML—meanwhile, the client also waits. Once the application has completed, it passes the dynamically-generated HTML back to the server, who then (finally) forwards it back to the user. This process of the server receiving a form, contacting an external application, receiving and returning the newly-generated HTML takes place through what is called the Web server's *Common Gateway Interface* (CGI). An overview of how CGI works is illustrated in [Figure 19-3](#), which shows you the execution and data flow, step-by-step from when a user submits a form until the resulting Web page is returned.

Figure 19-3. Overview of how CGI Works. CGI represents the interaction between a web server and the application which is required to process a user's form and generate the dynamic HTML that is eventually returned.



Forms input from the client sent to a Web server may include processing and perhaps some form of storage in a backend database. Just keep in mind that any time there are any

user-filled fields and/or a Submit button or image, it most likely involves some sort of CGI activity.

CGI applications which create the HTML are usually written in one of many higher-level programming languages which have the ability to accept user data, process it, and return value HTML back to the server. Today, these include: Perl, C, C++, or Python, to name a few. In this next section, we will look at how to create CGI applications in Python, with the help of the `cgimodule`.

CGI Applications

A CGI application is slightly different from a typical program. The primary differences are in the input, output, and user interaction aspects of a computer program.

When a CGI script starts, it will have the additional functionality of retrieving the user-supplied data, the input for the program comes from the data via the Web client, not a user on the server machine nor a disk file.

The output differs in that any data sent to standard output will be sent back to the connected Web client rather than to the screen, GUI window, or disk file. The data that is sent back must be a set of valid headers followed by HTML. If it is not and the Web client is a browser, an error (specifically, an Internal Server Error) will occur because Web clients such as browsers understand only valid HTTP data (i.e., MIME headers and HTML).

Finally, as you can probably guess, there is no user interaction with the script. All communication occurs among the Web client (on behalf of a user), the Web server, and the CGI application.

cgiModule

There is one primary class in the `cgi` module which does all the work: the `FieldStorage` class. This class should be instantiated when a Python CGI script begins, as it will read in all the pertinent user information from the Web client (via the Web server). Once this object has been instantiated, it will consist of a dictionary-like object which has a set of key-value pairs. The keys are the names of the form items that were passed in through the form while the values contain the corresponding data.

These values themselves can be one of three objects. They can be `FieldStorage` objects (instances) as well as instances of a similar class called `MiniFieldStorage`, which is used in cases where no file uploads or multiple part form data is involved.

`MiniFieldStorage` instances contain only the key-value pair of the name and the data.

Lastly, they can be a list of such objects. This occurs when a form contains more than one input item with the same field name.

For simple Web forms, you will usually find all `MiniFieldStorage` instances. All of our examples below pertain only to this general case.

Building CGI Application

Generating the Results Page

In [Example 19.2](#), we present the code for a simple Web form, `friends.htm`.

Example 19.2. Static Form Web Page (`friends.htm`)

This HTML file presents a form to the user with an empty field for the user's name and a set of radio buttons for the user to choose from.

```
<$nopage>
01 1 <HTML><HEAD><TITLE>
02 2 Friends CGI Demo (static screen) 003 3 </TITLE></HEAD>
004 4 <BODY><H3>Friends list for: <I>NEW USER</I></H3> 005 5 <FORM ACTION="/cgi-
bin/friends1.py">
06 6 <B>Enter your Name:</B>
07 7 <INPUT TYPE=text NAME=person SIZE=15>
08 8 <P><B>How many friends do you have?</B>
09 9 <INPUT TYPE=radio NAME=howmany VALUE="0"> CHECKED> 0
10 10 <INPUT TYPE=radio NAME=howmany VALUE="10"> 10
11 11 <INPUT TYPE=radio NAME=howmany VALUE="25"> 25
```

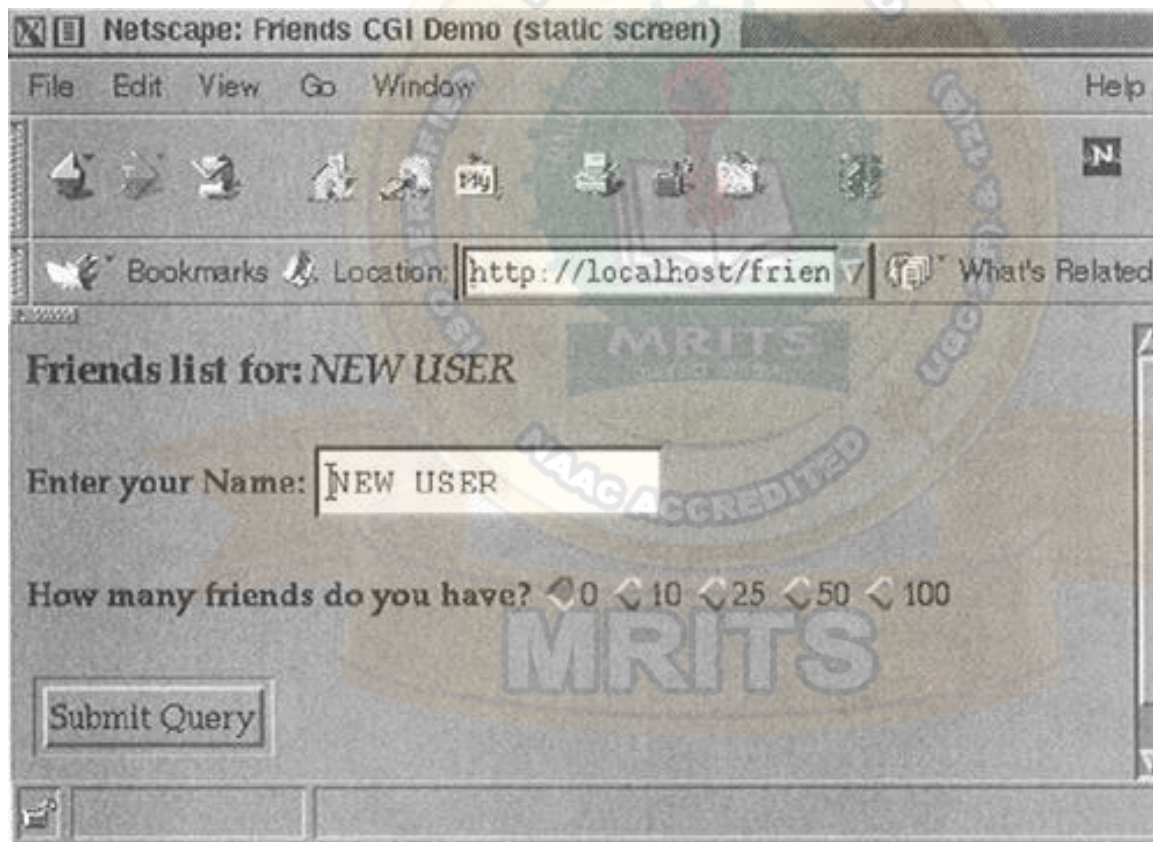
```
12 12 <INPUT TYPE=radio NAME=howmany VALUE="50"> 50
13 13 <INPUT TYPE=radio NAME=howmany VALUE="100"> 100
14 14 <P><INPUT TYPE=submit></FORM></BODY></HTML>
15 <$nopage>
```

As you can see in the code, the form contains two input variables: person and howmany. The values of these two fields will be passed to our CGI script, friends1.py.

You will notice in our example that we install our CGI script into the default cgi-bin directory (see the "Action" link) on the local host. (If this information does not correspond with your development environment, update the form action before attempting to test the Web page and CGI script.) Also, because a METHOD subtag is missing from the form action, all requests will be of the default type, GET. We choose the GET method because we do not have very many form fields, and also, we want our query string to show up in the "Location" (a.k.a. "Address," "Go To") bar so that you can see what URL is sent to the server.

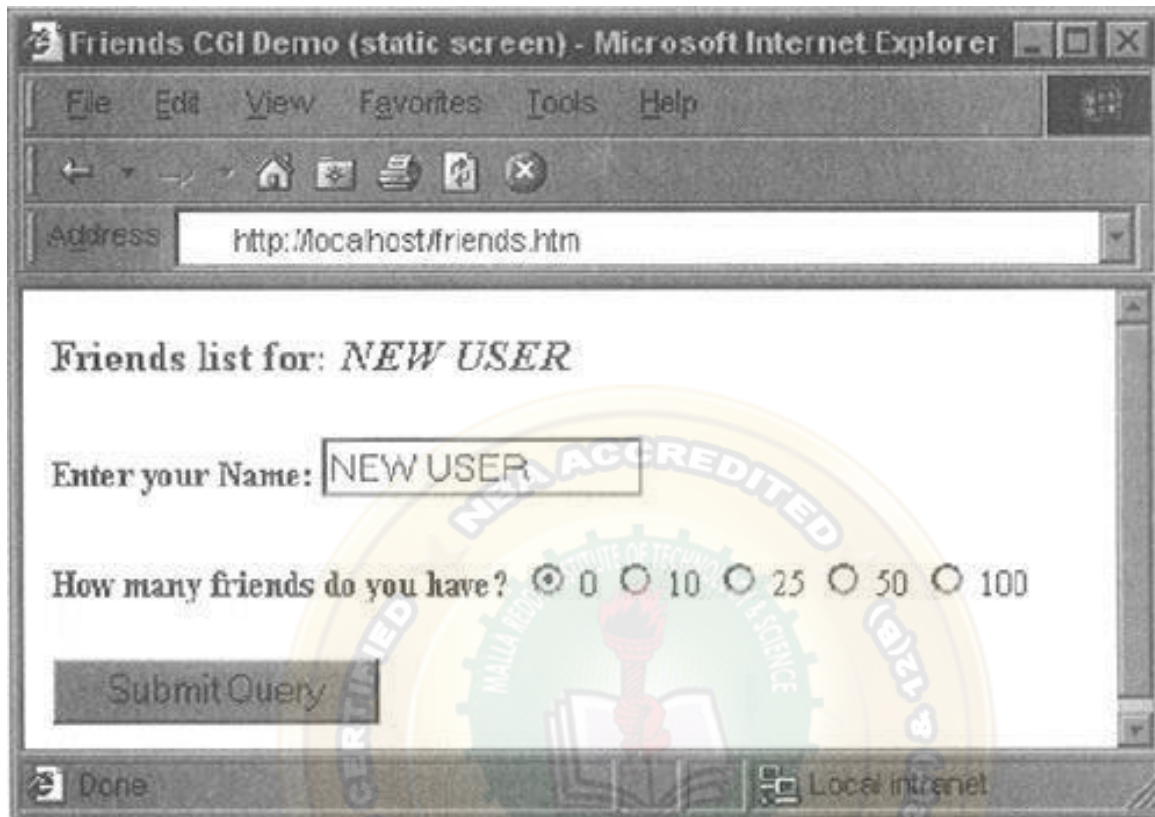
Let's take a look at the screen which is rendered by friends.htm in a Web browser. [Figure19-4](#) illustrates what the page would look like using Netscape Communicator 4 in a UNIX environment, while [Figure19-5](#) is an example of using Microsoft IE5 on Windows.

Figure 19-4. Friends Form Page in Netscape4 on Unix (friends.htm)



The input is entered by the user and the "Submit" button is pressed. (Alternatively, the user can also press the RETURN or Enter key within the text field to cause a similar effect.) When this occurs, the script in [Example 19.3](#), friends1.py, is executed via CGI.

Figure 19-5. Friends Form Page in IE5 on Windows (friends.htm)



Example 19.3. Results Screen CGI code (friends1.py)

This CGI script grabs the person and howmany fields from the form and uses that data to create the dynamically-generated results screen.

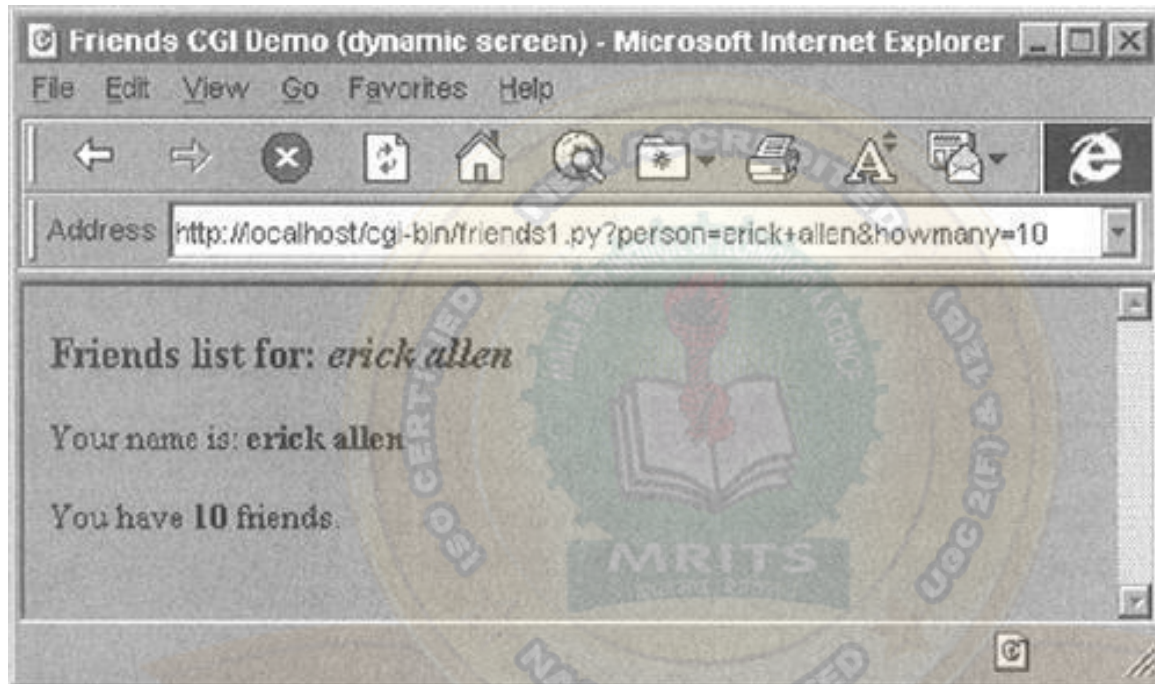
```
<$nopcode>
001 1    #!/usr/bin/env python
002 2
003 3 import cgi
004 4
05 5 reshtml = "Content-Type: text/html\n
06 6 <HTML><HEAD><TITLE>
07 7Friends CGI Demo (dynamic screen)
08 8    008 8  </TITLE></HEAD>
09 9<BODY><H3>Friends list for: <I>%s</I></H3>
10 10 Your name is: <B>%s</B><P>
11 11 You have <B>%s</B> friends.
012 12 </BODY></HTML>"
013 13
14 14 form = cgi.FieldStorage()
15 15 who = form['person'].value
16 16 howmany = form['howmany'].value
17 017 17 print reshtml % (who, who, howmany)
18 018 <$nopcode>
```


This script contains all the programming power to read the form input and process it, as well as return the resulting HTML page back to the user. All the "real" work in this script takes place in only four lines of Python code (lines 14–17).

The form variable is our FieldStorage instance, containing the values of the person and howmany fields. We read these into the Python whoand howmany variables, respectively. The reshtml variable contains the general body of HTML text to return, with a few fields filled in dynamically, the data just read in from the form.

One possible resulting screen appears in [Figure19-6](#), assuming the user typed in "erick allen" as the name and clicked on the "10 friends" radio button.

Figure 19-6. Friends Results Page in IE3 on Windows



The screen snapshot this time is represented by the older IE3 browser in a Windows environment.

If you are a Web site producer, you may be thinking, "Gee, wouldn't it be nice if I could automatically capitalize this person's name, especially if they forgot?" This can easily be accomplished using Python CGI. (And we shall do so soon!)

Notice how on a GET request that our form variables and their values are added to the form action URL in the "Address" bar. Also, did you observe that the title for the friends.htm page has the word "static" in it while the output screen from friends.py has the word "dynamic" in *its* title? We did that for a reason: to indicate that friends.htm file is a static text file while the results page is dynamically-generated. In other words, the HTML for the results page did not exist on disk as a text file; rather, it was generated by our CGI script and returned it as if it *was* a local file.

In our next example, we will bypass static files altogether by updating our CGI script to be somewhat more multifaceted.

Generating Form and Results Pages

We obsolete friends.html and merge it into friends2.py. The script will now generate both the form page as well as the results page. But how can we tell which page to generate? Well, if there is form data being sent to us, that means that we should be creating a results page. If we do not get any information at all, that tells us that we should generate a form page for the user to enter his or her data.

Our new friends2.pyscript is shown in [Example 19.4](#).

Example 19.4. Generating Form and Results Pages (friends2.py)

Both friends.html and friends1.py are merged together as friends2.py. The resulting script can now output both form and results pages as dynamically-generated HTML and has the smarts to know which page to output.

```
<$nopcode>
001 1 #!/usr/bin/env python
002 2
003 3 import cgi
004 4
005 5 header =      'Content-Type: text/html\n\n'
006 6
07 7 formhtml =      "<HTML> <HEAD><TITLE>
08 8 Friends CGI Demo</TITLE></HEAD>
09 9<BODY><H3>Friends list for: <I>NEW USER</I><</H3>
10 10 <FORM ACTION="/cgi-bin/friends2.py">
11 11 <B>Enter your Name:</B>
12 12 <INPUT TYPE=hidden NAME=action VALUE=edit>
13 13 <INPUT TYPE=text NAME=person SIZE=15>
14 14 <P><B>How many friends do you have?</B>
15 15 015 15 %s
016 16 <P><INPUT TYPE=submit></FORM></BODY></HTML>"
017 17
19 18 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
20 19 019 19
20 20 def showForm():
21 21     friends = "
022 22     for i in [0, 10, 25, 50, 100]:
023 23         checked = "
024 24         if i == 0:
25 25             checked = 'CHECKED'
26 26         friends = friends + fradio % \
27 27             (str(i), checked, str(i))
028 28
029 29     print header + formhtml % (friends)
030 30
31 31 reshtml = "<HTML><HEAD><TITLE>
32 32 Friends CGI Demo</TITLE></HEAD>
33 33 <BODY><H3>Friends list for: <I>%s</I><</H3>
34 34 Your name is: <B>%s</B><P>
35 35 You have <B>%s</B> friends.
036 36 </BODY></HTML>"
037 37
38 38 def doResults(who, howmany):
39 39     print header + reshtml % (who, who, howmany)
040 40
41 41 def process():
42 42     form = cgi.FieldStorage()
43 43     if form.has_key('person'):
```

```

44 44         who = form['person'].value
45 45         else: <$nepage>
46 46         who = 'NEW USER'
047 47
48 48         if form.has_key('howmany'):
49 49             howmany = form['howmany'].value
50 50         else: <$nepage>
51 51             howmany = 0
052 52
53 53         if form.has_key('action'):
54 54             doResults(who, howmany)
55 55         else: <$nepage>
56 56 showForm()
057 57
058 58 if_name == '_main_':
59 59 process()
60 <$nepage>

```

So what did we change in our script? Let's take a look at some of the blocks of code in this script.

Line-by-line explanation Lines 1 – 5

In addition to the usual start-up and module import lines, we separate the HTTP MIME header from the rest of the HTML body because we will use it for both types of pages (form page and results page) returned and don't want to duplicate the text. We will add this header string to the corresponding HTML body when it comes time for output to occur.

Lines 7 – 29

All of this code is related to the now-integrated friends.htm form page in our CGI script. We have a variable for the form page text, formhtml, and we also have a string to build the list of radio buttons, fradio. We could have duplicated this radio button HTML text as it is in friends.htm, but we wanted to show how we could use Python to generate more dynamic output—see the **for**-loop on lines 22–27.

The showForm() function has the responsibility of generating a form for user input. It builds a set of text for the radio buttons, merges those lines of HTML into the main body of formhtml, prepends the header to the form, and then returns the entire wad of data back to the client by sending the entire string to standard output.

There are a couple of interesting things to note about this code. The first is the "hidden" variable in the form called action, containing the value, "edit" on line 12. This field is the only way we can tell which screen to display (i.e., the form page or the results page). We will see this field come into play in lines 53–56.

Also, observe that we set the 0 radio button as the default by "checking" it within the loop that generates all the buttons. This will also allow us to update the layout of the radio buttons and/or their values on a single line of code (line 18) rather than over multiple lines of text. It will also offer some more flexibility in letting the logic determine which radio button is checked—see the next update to our script, friends3.pycoming up.

Now you may be thinking, "Why do we need an action variable when I could just as well be checking for the presence of person or howmany?" That is a valid question because yes, you could have just used personor howmanyin this situation.

However, the action variable is a more conspicuous presence, in as far as its name as well as what it does—the code is easier to understand. The person and howmany variables are used for their values while the actionvariable is used as a flag.

The other reason for creating action is that we will be using it again to help us determine which page to generate. In particular, we will need to display a form *with* the resence of a person variable (rather than a results page)—this will break your code if you are solely relying on there being a personvariable.

Lines 31 – 39

The code to display the results page is practically identical to that of friends1.py.

Lines 41 – 56

Since there are different pages which can result from this one script, we created an overall process() function to get the form data and decide which action to take. The main portion of process() will also look familiar to the main body of code in friends1.py. There are two major differences, however.

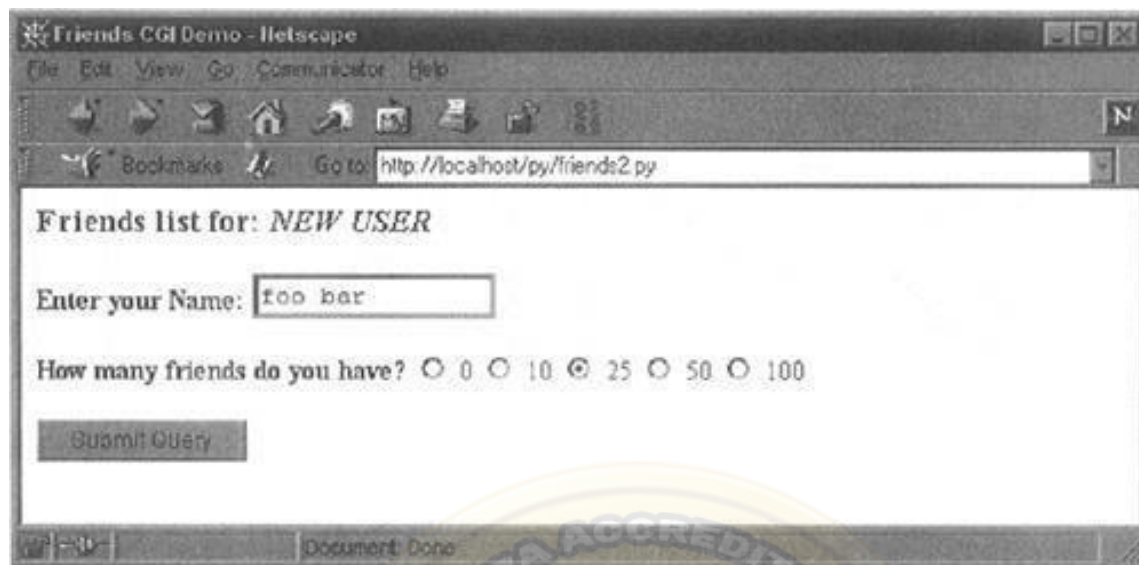
Since the script may or may not be getting the expected fields (invoking the script the first time to generate a form page, for example, will not pass any fields to the server), we need to "bracket" our retrieval of the form fields with if statements to check if they are even there. Also, we mentioned the action field above which helps us decide which page to bring up. The code that performs this determination is in lines 53–56.

In [Figure19-8](#) and [Figure19-7](#), you will see first the form screen generated by our script (with a name entered and radio button chosen), followed by the results page, also generated by our script.

Figure 19-8. Friends Form Page in Netscape4 on Windows



Figure 19-7. Friends Results Page in Netscape4 on Windows



If you look at the location or "Go to" bar, you will not see a URL referring to a static friends.htm file as you did in [Figure19-4](#) or [Figure19-5](#) earlier.

Fully Interactive Web Sites

Our final example will complete the circle. As in the past, a user enters his or her information from the form page. We then process the data and output a results page. Now we will add a link to the results page that will allow the user to go *back* to the form page, but rather than presenting a blank form, we will fill in the data that the user has already provided. We will also add some error processing to give you an example of how it can be accomplished.

We now present our final update, friends3.py in [Example 19.5](#).

Example 19.5. Full User Interaction and Error Processing (friends3.py)

By adding a link to return to the form page with information already provided, we have come "full circle," giving the user a fully-interactive Web surfing experience. Our application also now performs simple error checking which notifies the user if no radio button was selected.

```
<$npage>
001 1 #!/usr/bin/env python
002 2
03 3 import cgi
04 4 from urllib import quote_plus
05 5 from string import capwords
006 6
07 7 header = 'Content-Type: text/html\n\n'
08 8 url = '/cgi-bin/friends3.py'
009 9
10 10 errhtml = "<HTML><HEAD><TITLE>
11 11 Friends CGI Demo</TITLE></HEAD>
12 12 <BODY><H3>ERROR</H3>
013 13 <B>%s</B><P>
14 14 <FORM><INPUT TYPE=button VALUE=Back
15 15 ONCLICK="window.history.back()"</FORM <$npage>
016 16 </BODY></HTML>"
017 17
18 18 def showError(error_str):
19 19     print header + errhtml % (error_str)
20 20
21 21 formhtml = "<HTML><HEAD><TITLE>
22 22 Friends CGI Demo</TITLE></HEAD>
23 23 <BODY><H3>Friends list for: <I>%s</I></H3>
24 24 <FORM ACTION="%s">
25 25 <B>Your Name:</B>
26 26 <INPUT TYPE=hidden NAME=action VALUE=edit>
27 27 <INPUT TYPE=text NAME=person VALUE="%s" SIZE=15>
28 28 <P><B>How many friends do you have?</B>
29 29 029 29 %s
030 30 <P><INPUT TYPE=submit></FORM></BODY></HTML>"
031 31
032 32 fradio = '<INPUT TYPE=radio NAME=howmany VALUE="%s" %s> %s\n'
033 33
```

```

34 34 def showForm(who, howmany):
35 35     friends = "
036 36     for i in [0, 10, 25, 50, 100]:
37 37         checked = "
38 38         if str(i) == howmany:
39 39             checked = 'CHECKED'
40 40             friends = friends + fradio % \
41 41             (str(i), checked, str(i))
042 42     print header + formhtml % (who, url, who, friends)
043 43
44 44 reshtml = "<HTML><HEAD><TITLE>
45 45 Friends CGI Demo</TITLE></HEAD>
46 46 <BODY><H3>Friends list for: <I>%s</I></H3>
47 47 Your name is: <B>%s</B><P>
48 48 You have <B>%s</B> friends.
49 49 <P>Click <A HREF="%s">here</A> to edit your data again.
50 50 </BODY></HTML>"
051 51
52 52 def doResults(who, howmany):
53 53     newurl = url + '?action=reedit&person=%s&howmany=%s' \
54 54         (quote_plus(who), howmany)
055 55     print header + reshtml % (who, who, howmany, newurl)
056 56
57 57 def process():
58 58     error = "
59 59     form = cgi.FieldStorage()
060 60
61 61     if form.has_key('person'):
62 62         who = capwords(form['person'].value)
63 63     else: <$npage>
64 64         who = 'NEW USER'
065 65
66 66     if form.has_key('howmany'):
67 67         howmany = form['howmany'].value
68 68     else: <$npage>
69 69         if form.has_key('action') and \
70 70             form['action'].value == 'edit':
71 71             error = 'Please select number of friends.'
72 72         else: <$npage>
073 73             howmany = 0
074 74
75 75     if not error:
76 76         if form.has_key('action') and \
77 77             form['action'].value != 'reedit':
78 78             doResults(who, howmany)
79 79     else: <$npage>
80 80         showForm(who, howmany)
81 81     else: <$npage>
82 82         showError(error)

```

```
083 83
084 84 if _name == '_main_':
85 85     process()
86 <$nopage>
```

friends3.py is not too unlike friends2.py. We invite the reader to compare the differences; we present a brief summary of the major changes for you here:

Abridged line-by-line explanation Line 8

We take the URL out of the form because we now need it in two places, the results page being the new customer.

Lines 10 – 19, 69 – 71, 75 – 82

All of these lines deal with the new feature of having an error screen. If the user does not select a radio button indicating the number of friends, the howmany field is not passed to the server. In such a case, the showError() function returns the error page to the user.

The error page also features a JavaScript "Back" button. Because buttons are input types, we need a form, but no action is needed because we are simply just going back one page in the browsing history. Although our script currently supports (a.k.a. detects, tests for) only one type of error, we still use a generic error variable in case we wanted to continue development of this script to add more error detection in the future.

Lines 27, 38–41, 49, and 52–55

One goal for this script is to create a meaningful link back to the form page from the results page. This is implemented as a link to give the user the ability to return to a form page to update the data her or she entered, in case it was erroneous. The new form page makes sense only if it contains information pertaining to the data that has already been entered by the user. (It is frustrating for users to reenter their information from scratch!)

To accomplish this, we need to embed the current values into the updated form. In line 27, we add a value for the name. This value will be inserted into the name field, if given. Obviously, it will be blank on the initial form page. In Line 38–41, we set the radio box which corresponds to the number of friends currently chosen. Finally, on lines 49 and the updated doResults() function on lines 52–55, we create the link with all the existing information which "returns" the user to our modified form page.

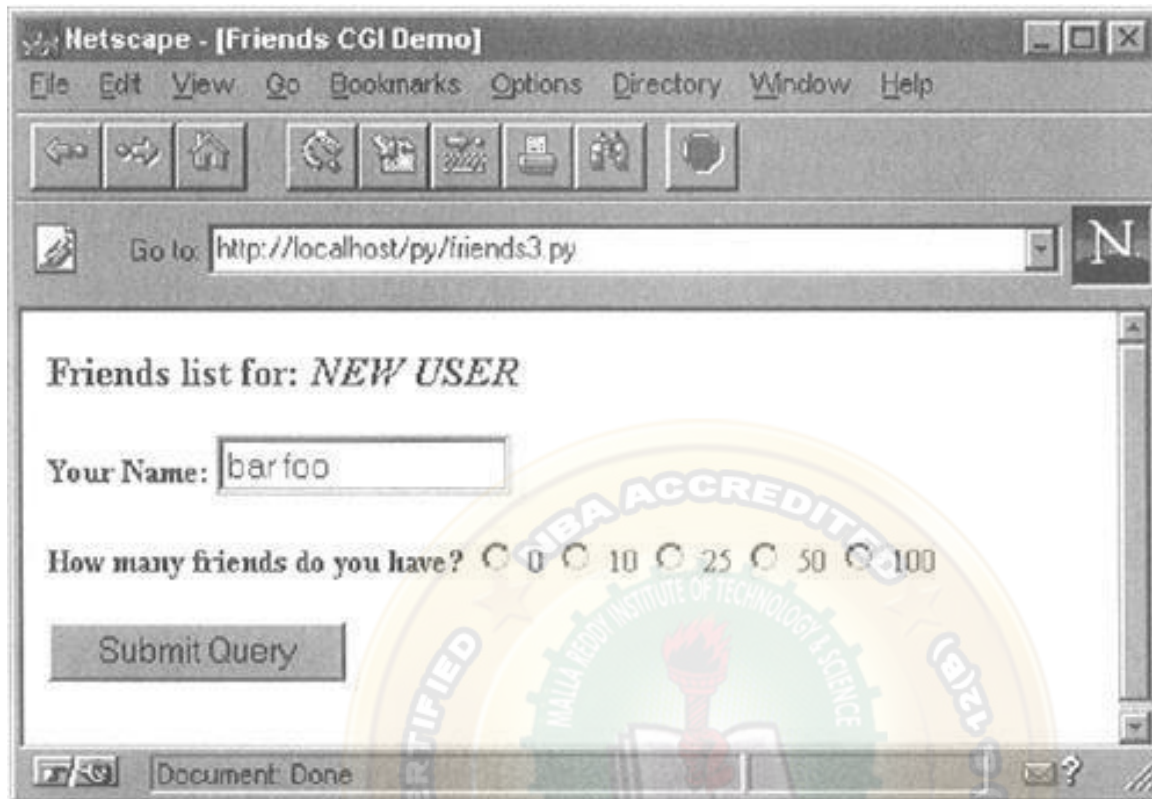
Line 62

Finally, we added a simple feature that we thought would add a nice aesthetic touch. In the screens for friends1.py and friends2.py, the text entered by the user as his or her name is taken verbatim. You will notice in the screens above that if the user does not capitalize his or her names, that is reflected in the results page. We added a call to the string.capitalize() function to automatically capitalize a user's name. The capitalize() function will capitalize the first letter of each word in the string that is passed in. This may or may not be a desired feature, but we thought that we would share it with you so that you know that such functionality exists.

We will now present four screens which shows the progression of user interaction with this CGI form and script.

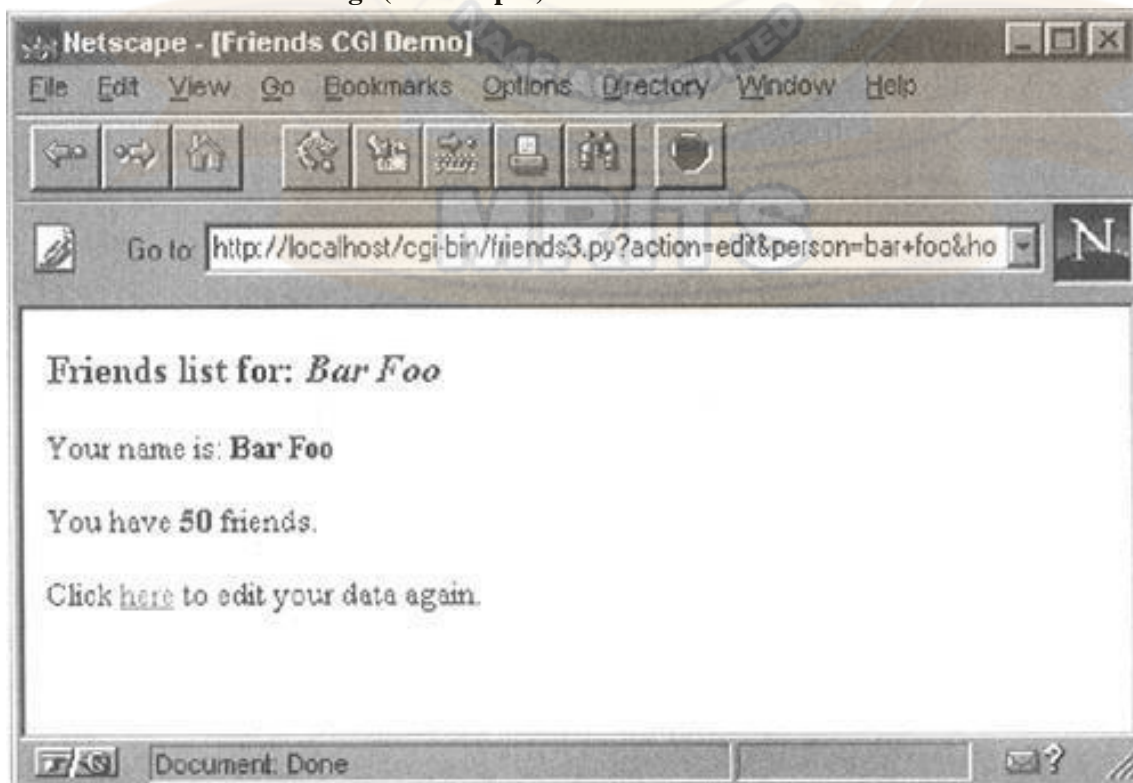
In the first screen, shown in [Figure19-9](#), we invoke friends3.py to bring up the now-familiar form page. We enter a name "bar foo," but deliberately avoid checking any of the radio buttons. The resulting error after submitting the form can be seen in the second screen ([Figure19-10](#)).

Figure 19-9. Friends Initial Form Page in Netscape3 on Windows



We click on the "Back" button, check the "50" radio button, and resubmit our form. The results page, seen in [Figure 19-11](#), is also familiar, but now has an extra link at the bottom. This link will take us back to the form page.

Figure 19-11. Friends Results Page (Valid Input)



The only difference between the new form page and our original is that all the data filled in by the user is now set as the "default" settings, meaning that the values are already available in the form. We can see this in [Figure19-12](#).

Figure 19-10. Friends Error Page (invalid user input)

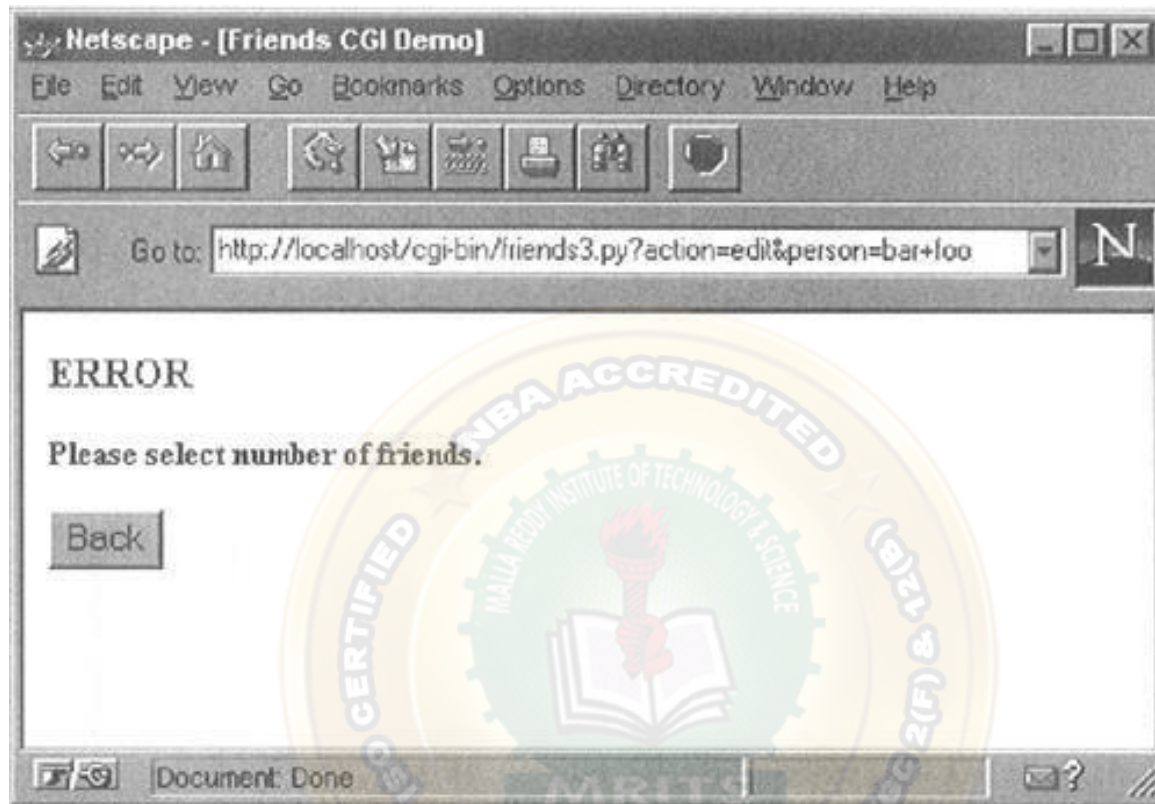
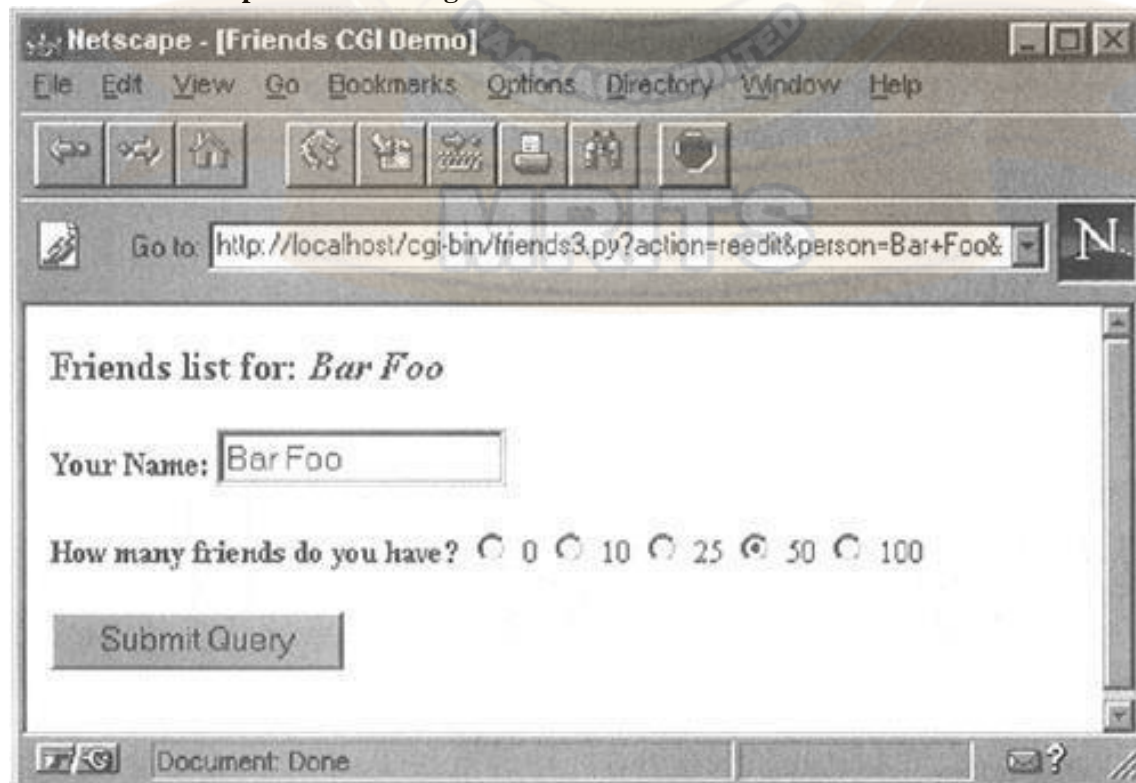


Figure 19-12. Friends Updated Form Page with Current Information



Now the user is able to make changes to either of the fields and resubmit their form.

You will no doubt begin to notice that as our forms and data get more complicated, so does the generated HTML, especially for complex results pages. If you ever get to a point where generating the HTML text is interfering with your application, you may consider connecting with a Python module such as HTMLgen, an external Python module which specializes in HTML generation.

Advanced CGI

We will now take a look at some of the more advanced aspects of CGI programming. These include: the use of *cookies*—cached data saved on the client side, multiple values for the same CGI field and file upload using multipart form submissions. To save space, we will show you all three of these features with a single application. Let's take a look at multipart submissions first.

Multipart Form Submission and File Uploading

Currently, the CGI specifications only allow two types of form encodings, "application/x-www-form-urlencoded" and "multipart/form-data." Because "application/x-www-form-urlencoded" is the default, there is never a need to state the encoding in the FORM tag like this:

```
<FORM enctype="application/x-www-form-urlencoded" ...>
```

But for multipart forms, you must explicitly give the encoding as:

```
<FORM enctype="multipart/form-data" ...>
```

You can use either type of encoding for form submissions, but at this time, file uploads can only be performed with the multipart encoding. Multipart encoding was invented by Netscape in the early days but since has been adopted by Microsoft (starting with version 4 of Internet Explorer) as well as other browsers.

File uploads are accomplished using the file input type:

```
<INPUT type=file name=...>
```

This directive presents an empty text field with a button on the side which allows you to browse your file directory structure for a file to upload. On most browsers, this button says "Browse," but your mileage may vary. (For example, we will be using the Opera browser in our examples which has a button labeled with ellipses "...".)

When using multipart, your Web client's form submission to the server will look amazingly like (multipart) e-mail messages with attachments. A separate encoding was needed because it just wouldn't be necessarily wise to "urlencode" a file, especially a binary file. The information still gets to the server, but is just "packaged" in a different way.

Regardless of whether you use the default encoding or the multipart, the cgi module will process them in the same manner, providing keys and corresponding values in the form submission. You will simply access the data through your FieldStorage instance as before.

Multivalued Fields

In addition for file uploads, we are also going to show you how to process fields with multiple values. The most common case is when you have a set of checkboxes allowing a user to select from various choices. Each of the checkboxes is labeled with the same field name, but to differentiate them, each will have a different value associated with a particular checkbox.

As you know, the data from the user is sent to the server in key-value pairs during form submission. When more than one checkbox is submitted, you will have multiple values associated with the same key. In these cases, rather than being given a single MiniFieldStorage instance for your data, the cgi module will create a list of such instances which

you will iterate over to obtain the different values. Not too painful at all.

Cookies

Finally, we will use cookies in our example. If you are not familiar with cookies, they are just bits of data information which a server at a Web site will request to be saved on the client side, e.g., the browser.

Because HTTP is a "stateless" protocol, information that has to be carried from one page to another can be accomplished by using key-value pairs in the request as you have seen in the GET requests and screens earlier in this chapter. Another way of doing it, as we have also seen before, is using hidden form fields, such as the action variable in some of the later friends*.py scripts. These variables and their values are managed by the server because the pages they return to the client must embed these in generated pages.

One alternative to maintaining persistency in state across multiple page views is to save the data on the client side instead. This is where cookies come in. Rather than embedding data to be saved in the returned Web pages, a server will make a request to the client to save a cookie. The cookie is linked to the domain of the originating server (so a server cannot set nor override cookies from other Web sites) and has an expiration date (so your browser doesn't become cluttered with cookies).

These two characteristics are tied to a cookie along with the key-value pair representing the data item of interest. There are other attributes of cookies such as a domain subpath or a request that a cookie should only be delivered in a secure environment.

By using cookies, we no longer have to pass the data from page to page to track a user. Although they have been subject to a good amount of controversy over the privacy issue, most Web sites use cookies responsibly. To prepare you for the code, a Web server requests a client store a cookie by sending the "Set-Cookie" header immediately before the requested file.

Once cookies are set on the client side, requests to the server will automatically have those cookies sent to the server using the HTTP_COOKIE environment variable. The cookies are delimited by semicolons and come in "key=value" pairs. All your application needs to do to access the data values is to split the string several times (i.e., using string.split() or manual parsing). The cookies are delimited by semicolons (;), and each key-value pair is separated by equal signs (=).

Like multipart encoding, cookies originated from Netscape, who implemented cookies and wrote up the first specification which is still valid today. You can access this document at the following Web site:

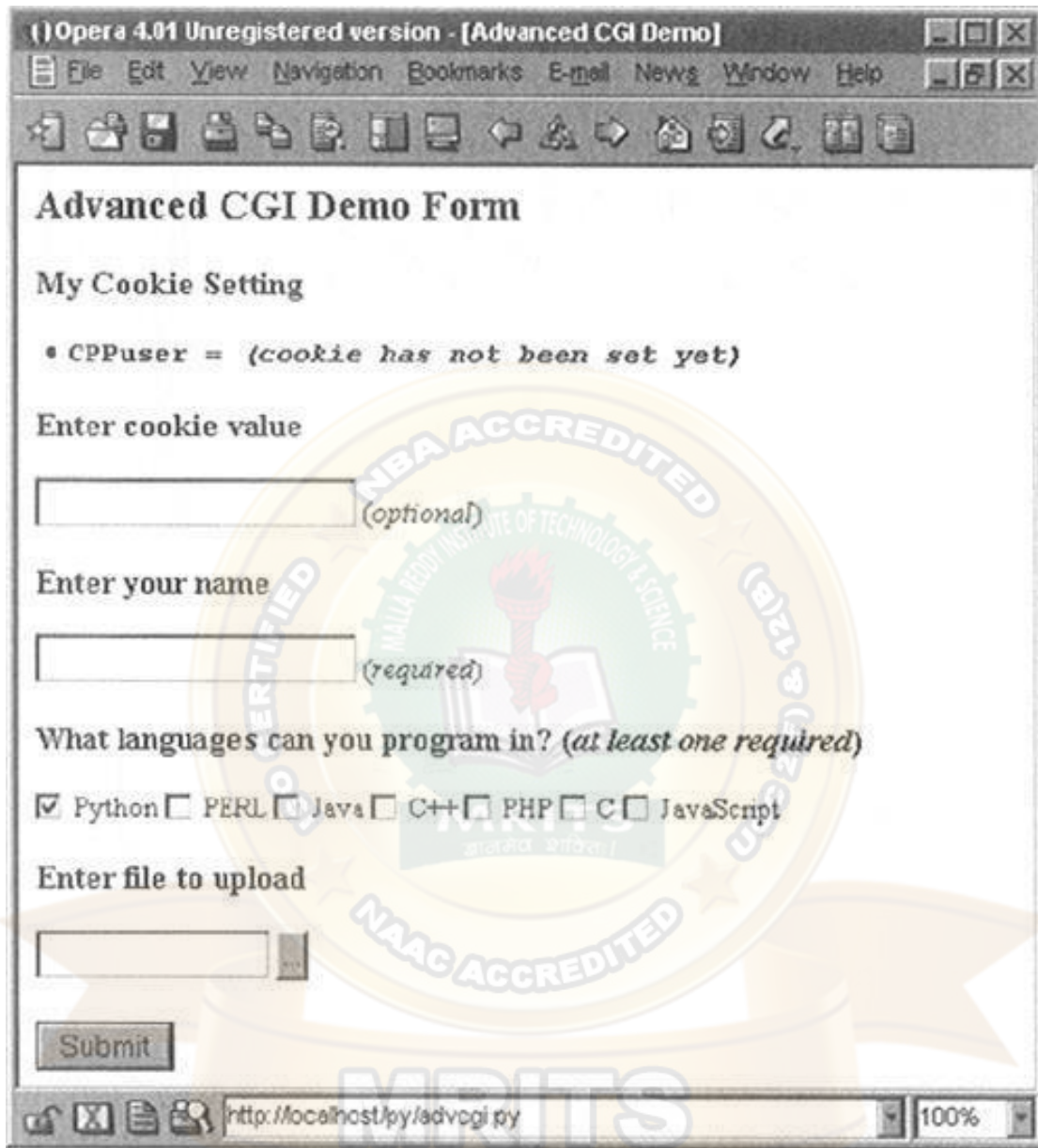
http://www.netscape.com/newsref/std/cookie_spec.html

Once cookies are standardized and this document finally obsoleted, you will be able to get more current information from Request for Comment documents (RFCs). The most current one for cookies at the time of publication is RFC 2109.

Using Advanced CGI

We now present our CGI application, advcgi.py, which has code and functionality not too unlike the friends3.py script seen earlier in this chapter. The default first page is a user fill-out form consisting of four main parts: user-set cookie string, name field, checkbox list of programming languages, and file submission box. An image of this screen can be seen in [Figure19-13](#), this time using the Opera 4 browser in a Windows environment.

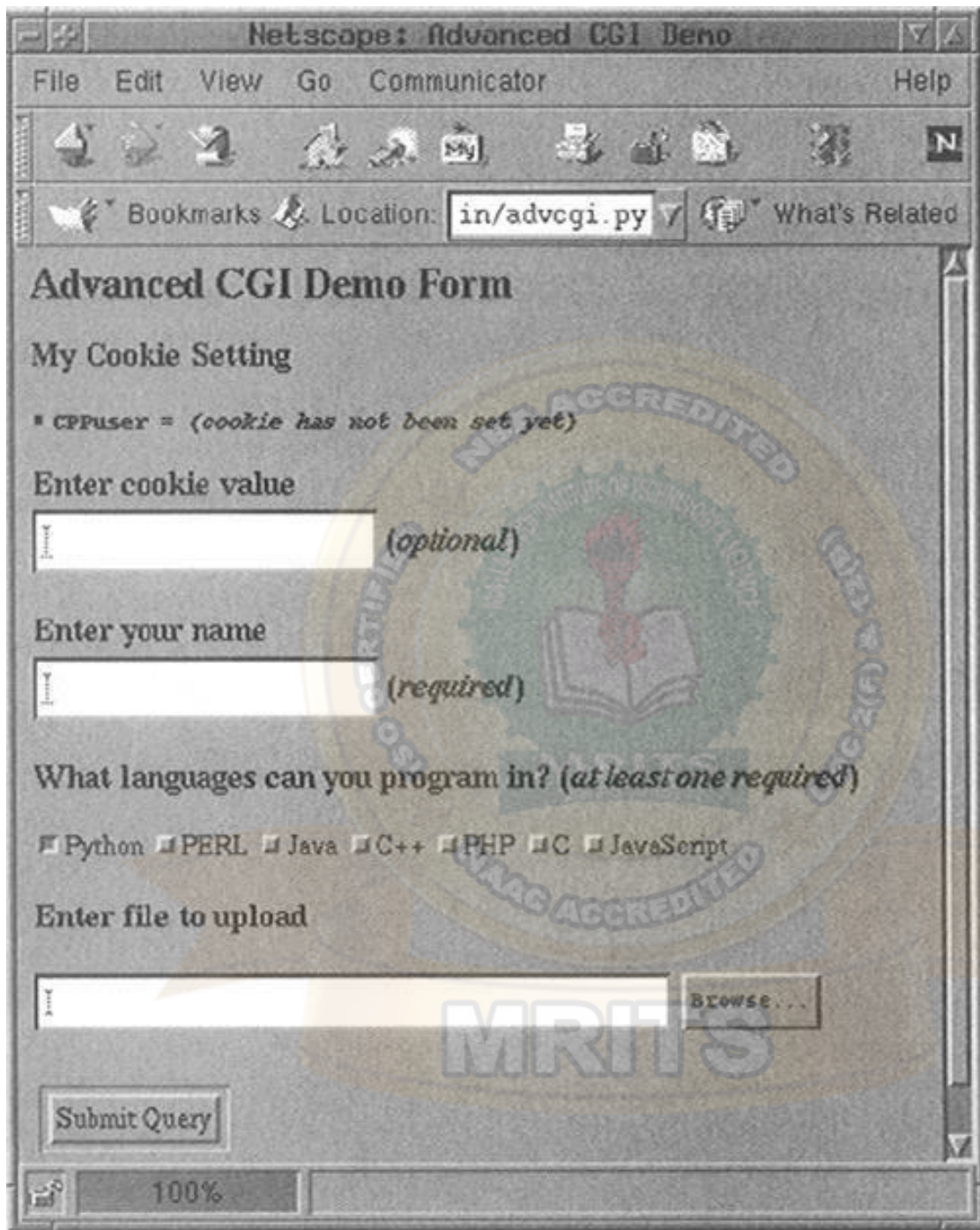
Figure 19-13. Upload and Multivalue Form Page in Opera4 on Windows



In a browser world dominated by the Netscape and Microsoft browsers, we seldom hear of others such as Opera and Lynx, but they are out there! Opera, in particular, is known to have excellent footprint (memory size) and speed characteristics.

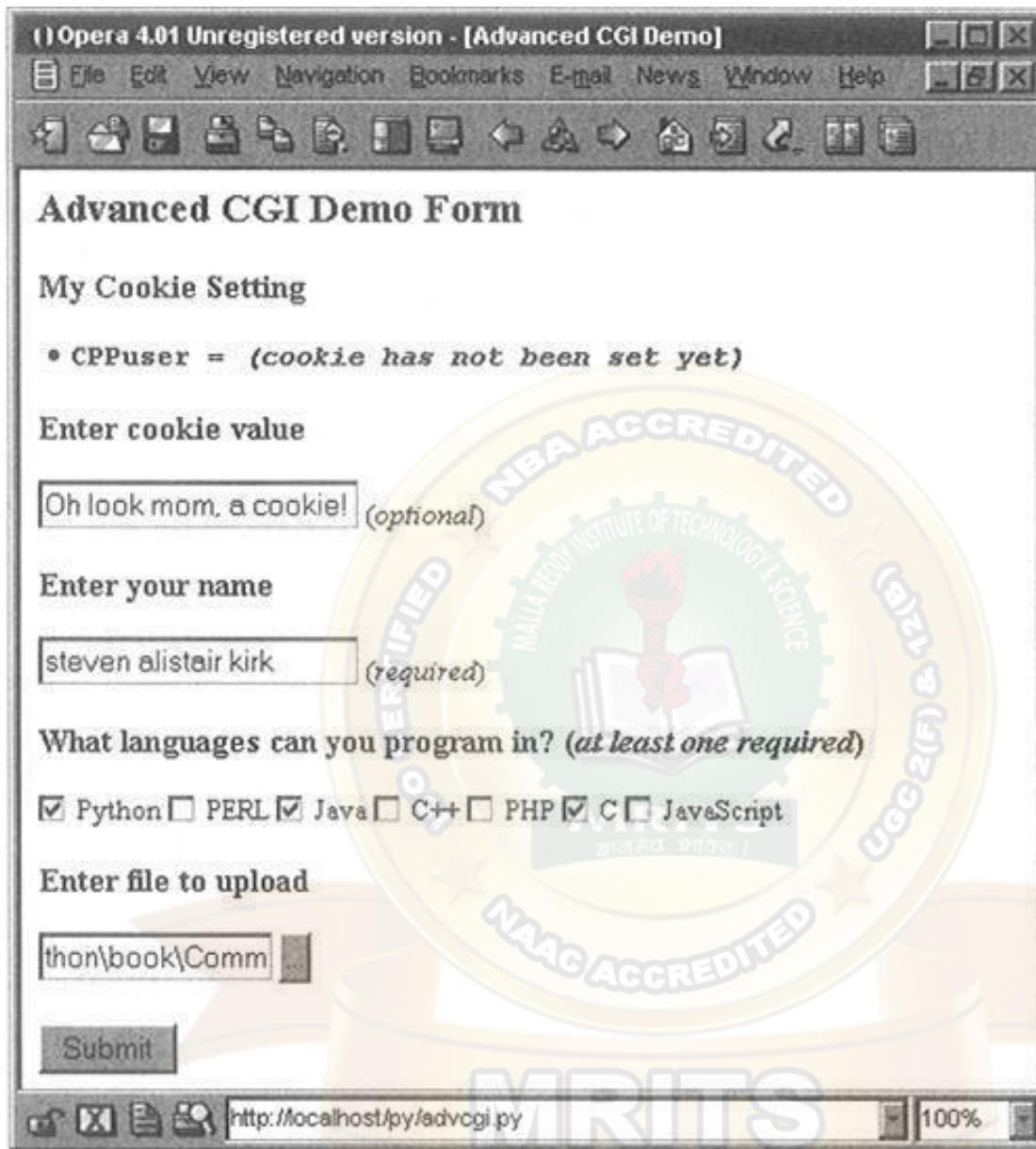
Well, just so you aren't totally uncomfortable, let's take a peek at what the same form looks like from Netscape running on Linux, as in [Figure19-14](#). As you can see, Netscape uses "Browse" as the file upload label instead of the ellipses. (The rest of the screens for this section will feature Opera.)

Figure 19-14. The Same Advanced CGI Form but in Netscape4 on Linux



From this form, we can enter our information, such as the sample data given in [Figure19-15](#).

Figure 19-15. One Possible Form Submission in our Advanced CGI Demo

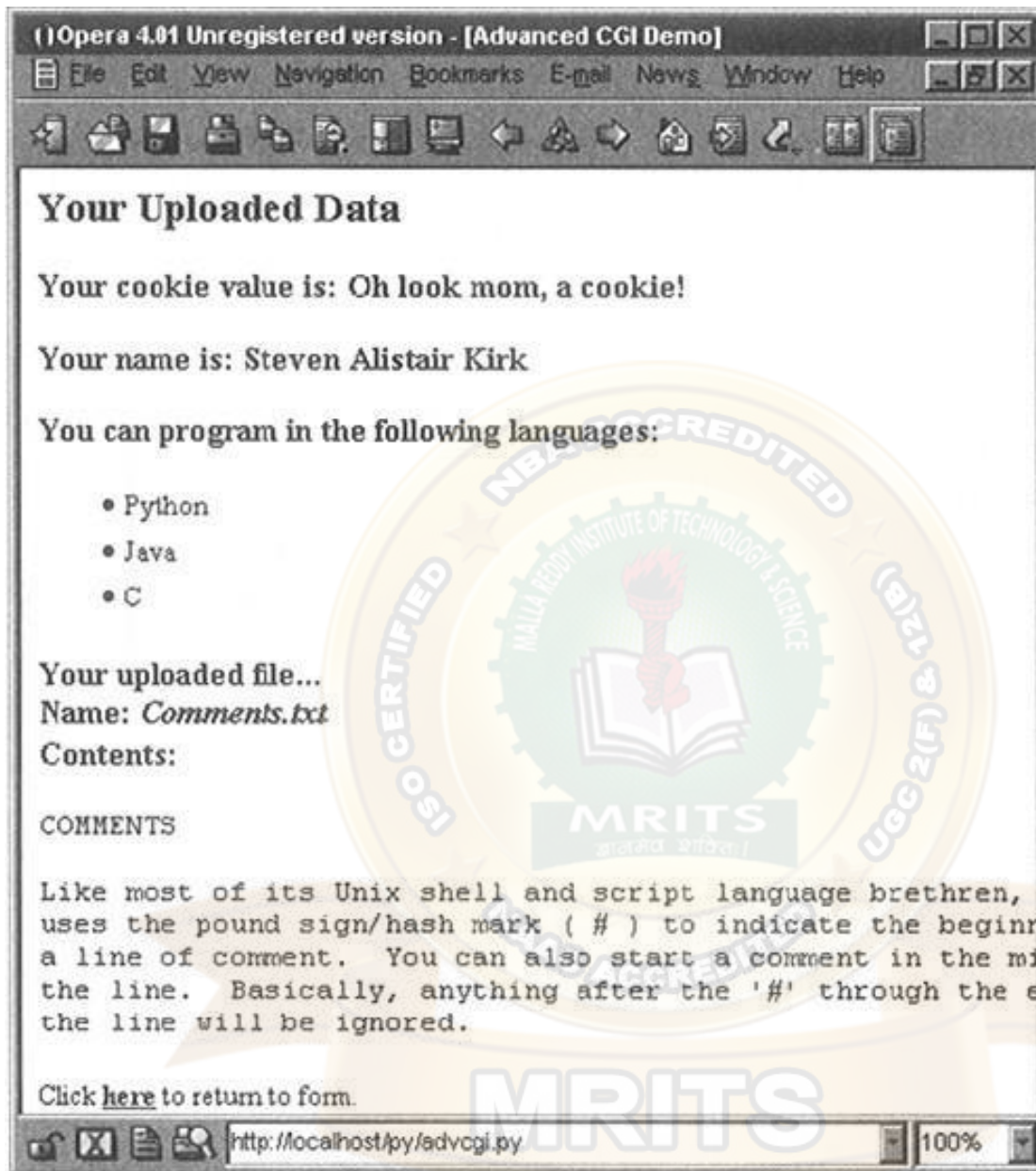


The data is submitted to the server using multipart encoding and is retrieved in the same manner on the server side using the FieldStorage instance. The only tricky part is in retrieving the uploaded file. In our application, we choose to iterate over the file, reading it line-by-line. It is also possible to read in the entire contents of the file if you are not wary of its size.

Since this is the first occasion data is received by the server, it is at this time, when returning the results page back to the client, that we use the "Set-Cookie:" header to cache our data in browser cookies.

In [Figure19-16](#), you will see the results after submitting our form data. All the fields the user entered are shown on the page. The contents of the filename given in the final dialog box was actually uploaded to the server and displayed as well.

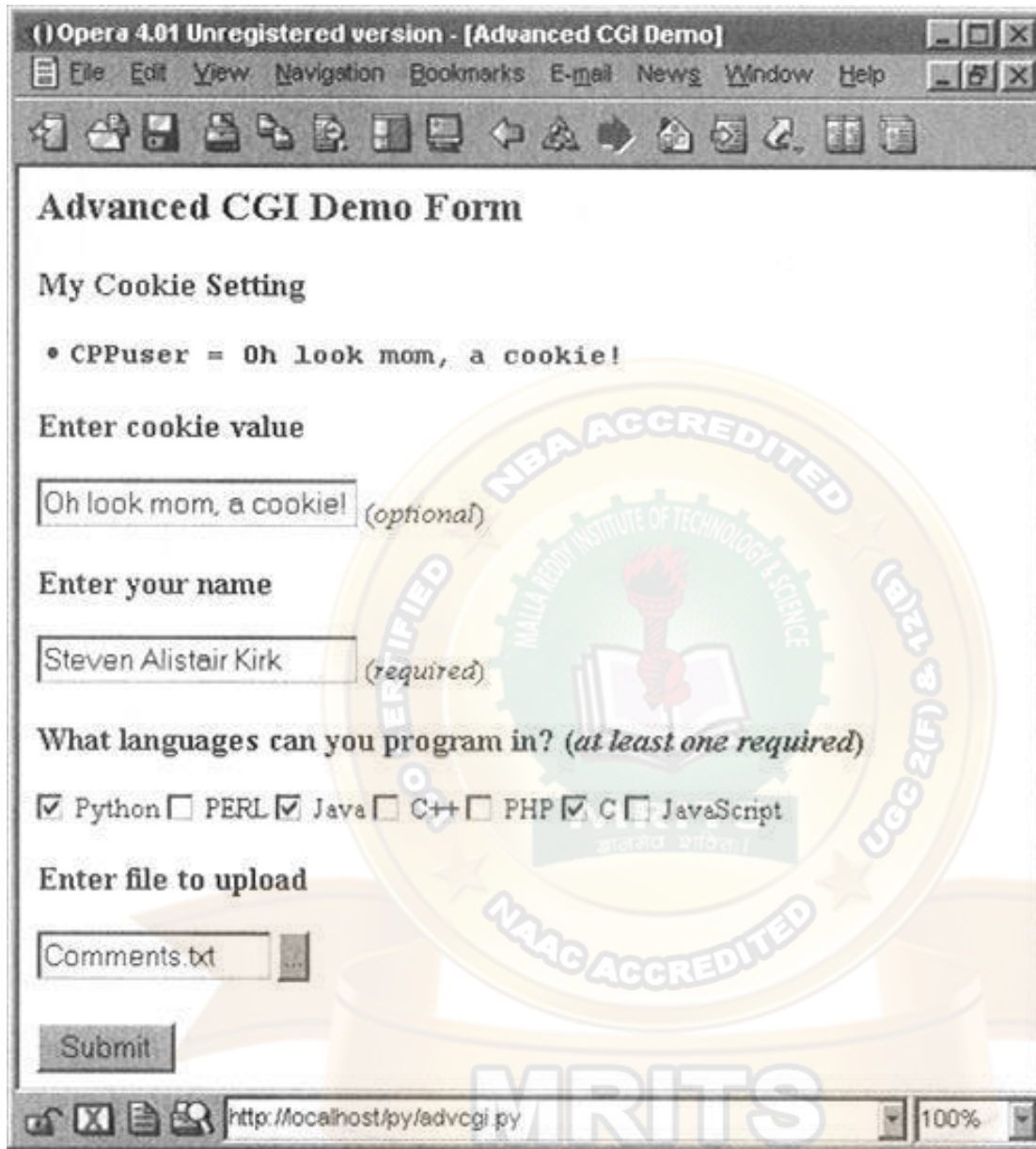
Figure 19-16. Results Page Generated and Returned by the Web Server



You will also notice the link at the bottom of the results page which returns us to the form page, again using the same CGI script.

If we click on that link at the bottom, no form data is submitted to our script, causing a form page to be displayed. Yet, as you can see from [Figure 19-17](#), what shows up is anything but an empty form! Information previously entered by the user shows up! How did we accomplish this with no form data (either hidden or as query arguments in the URL)? The secret is that the data is stored on the client side in cookies, two in fact.

Figure 19-17. Form Page With Data Loaded from the Client Cookies



The user cookie holds the string of data typed in by the user in the "Enter cookie value" form field, and the user's name, languages they are familiar with, and uploaded file are stored in the info cookie.

When the script detects no form data, it shows the form page, but before the form page has been created, it grabs the cookies from the client (which are automatically transmitted by the client when the user clicks on the link) and fills out the form accordingly. So when the form is finally displayed, all the previously entered information appears to the user like magic.

We are sure you are eager to take a look at this application, so here it is, presented in [Example 19-6](#).

Example 19.6. Advanced CGI Application (advcgi.py)

The crawler has one main class which does everything, AdvCGI. It has methods to show either form, error, or results pages as well as those which read or write cookies from/to the client (a web browser).

```
<$nopage>
001 1      #!/usr/bin/env python
002 2
```

```

003 3  from      cgi import FieldStorage
004 4  from      os import environ
005 5  from      cStringIO import StringIO
006 6  from      urllib import quote, unquote
007 7  from string import capwords, strip, split, join 008 8
009 9  class AdvCGI:
010 10
11 11      header = 'Content-Type: text/html\n\n'
12 12      url = '/py/advcgi.py'
013 13
14 14      formhtml = "<HTML><HEAD><TITLE>
15 15      Advanced CGI Demo</TITLE></HEAD>
16 16      <BODY><H2>Advanced CGI Demo Form</H2>
17 17      <FORM METHOD=post ACTION="%s ENCTYPE="multipart/form-data">
18 18      <H3>My Cookie Setting</H3>
19 19      <LI> <CODE><B>CPPuser = %s</B></CODE>
20 20      <H3>Enter cookie value<BR>
21 21      <INPUT NAME=cookie value="%s"> (<I>optional</I></H3>
22 22      <H3>Enter your name<BR>
23 23      <INPUT NAME=person VALUE="%s"> (<I>required</I></H3>
24 24 024 24 <H3>What languages can you program in?
025 25 (<I>at least one required</I></H3>
026 26 %s
27 27 <H3>Enter file to upload</H3>
28 28 <INPUT TYPE=file NAME=upfile VALUE="%s" SIZE=45>
29 29 <P><INPUT TYPE=submit>
030 30 </FORM></BODY></HTML>"
031 31
32 32      langSet = ('Python', 'PERL', 'Java', 'C++', 'PHP',
33 33                'C', 'JavaScript')
34 34      langItem = \
35 35          35 <INPUT TYPE=checkbox NAME=lang VALUE="%s"%s> %s\n'
36 36          036 36
037 37      def getPPCCookies(self):# read cookies from client
038 38          if environ.has_key('HTTP_COOKIE'):
39 39              for eachCookie in map(strip, \
40 40                  split(environ['HTTP_COOKIE'], ';')):
41 41                  if len(eachCookie) > 6 and \
42 42                      eachCookie[:3] == 'CPP':
43 43                      tag = eachCookie[3:7]
44 44                      try: <$nopage>
45 45                          self.cookies[tag] = \
46 46                              eval(unquote(eachCookie[8:]))
47 47                      except (NameError, SyntaxError):
48 48                          self.cookies[tag] = \
49 49                              unquote(eachCookie[8:])
50 50          else: <$nopage>
51 51              51 self.cookies['info'] = self.cookies['user'] = "
52 52              052 52

```

```

53 53         if self.cookies['info'] != ":
54 54             self.who, langStr, self.fn = \
55 55                 split(self.cookies['info'], ':')
56 56             self.langs = split(langStr, ',')
57 57         else: <$nopage>
58 58             self.who = self.fn = '
59 59             self.langs = ['Python']
060 60
061 61         def showForm(self):                               # show fill-out form
062 62             self.getCPPCookies()
063 63             langStr = "
064 64             for eachLang in AdvCGI.langSet:
065 65                 if eachLang in self.langs:
066 66                     langStr = langStr + AdvCGI.langItem % \
067 67                     (eachLang, 'CHECKED', eachLang)
068 68                 else: <$nopage>
069 69                     langStr = langStr + AdvCGI.langItem % \
070 70                             (eachLang, ", eachLang)
071 71
072 72                 if not self.cookies.has_key('user') or \
073 73                     self.cookies['user'] == ":
74 74 cookStatus = '<I>(cookie has not been set yet)</I>'
75 75 userCook = "
76 76 else: <$nopage>
77 77             userCook = cookStatus = self.cookies['user']
78 78
079 79             print AdvCGI.header + AdvCGI.formhtml % (AdvCGI.url,
080 80                 cookStatus, userCook, self.who, langStr, self.fn)
081 81
82 82 errhtml = "<HTML><HEAD><TITLE>
83 83 Advanced CGI Demo</TITLE></HEAD>
84 84 <BODY><H3>ERROR</H3>
085 85     <B>%s</B><P>
86 86 <FORM><INPUT TYPE=button VALUE=Back
87 87 ONCLICK="window.history.back()"></FORM>
088 88     </BODY></HTML>"
089 89
90 90 def showError(self):
91 91         print AdvCGI.header + AdvCGI.errhtml % (self.error)
92 92
093 93         reshtml =         "<HTML><HEAD><TITLE>
094 94 Advanced CGI Demo</TITLE></HEAD>
095 95 <BODY><H2>Your Uploaded Data</H2>
096 96 <H3>Your cookie value is: <B>%s</B></H3>
097 97 <H3>Your name is: <B>%s</B></H3>
098 98 <H3>You can program in the following languages:</H3>
099 99 <UL>%s</UL>
100 100 <H3>Your uploaded file...<BR>

```

```

101 101  Name: <I>%s</I><BR>
102 102  Contents:</H3>
103 103  <PRE>%s</PRE>
104 104  Click <A HREF="%s"><B>here</B></A> to return to form.
105 105  </BODY></HTML>"
106 106
107 107  def setCPPCookies(self):# tell client to store cookies
108 108  or eachCookie in self.cookies.keys():
109 109  print 'Set-Cookie: CPP%s=%s; path=' % \
110 110  (eachCookie, quote(self.cookies[eachCookie]))
111 111
112 112  def doResults(self):# display results page
113 113  MAXBYTES = 1024
114 114  langlist = "
115 115  for eachLang in self.langs:
116 116  langlist = langlist + '<LI>%s<BR>' % eachLang
117 117
118 118  filedata = "
119 119  while len(filedata) < MAXBYTES:# read file chunks
120 120  data = self.fp.readline()
121 121  if data == "": break <$npage>
122 122  filedata = filedata + data
123 123  else: # truncate if too long
124 124  filedata = filedata + \
125 125  '... <B><I>(file truncated due to size)</I></B>' <$npage>
126 126  self.fp.close()
127 127  if filedata == "":
128 128  filedata = \
129 129  '<B><I>(file upload error or file not given)</I></B>' <$npage>
130 130  filename = self.fn
131 131
132 132  if not self.cookies.has_key('user') or \
133 133  self.cookies['user'] == "":
134 134  cookStatus = '<I>(cookie has not been set yet)</I>' <$npage>
135 135  userCook = "
136 136  else: <$npage>
137 137  userCook = cookStatus = self.cookies['user']
138 138
139 139  self.cookies['info'] = join([self.who, \
140 140  join(self.langs, '), filename], ':')
141 141  self.setCPPCookies()
142 142  print AdvCGI.header + AdvCGI.reshtml % \
143 143  (cookStatus, self.who, langlist,
144 144  filename, filedata, AdvCGI.url)
145 145
146 146  def go(self):# determine which page to return
147 147  self.cookies = {}
148 148  self.error = "
149 149  form = FieldStorage()

```



```

150 150      if form.keys() == []:
151 151          self.showForm()
152          152      return <$npage>
153 153          153 153
154 154      if form.has_key('person'):
155 155          self.who = capwords(strip(form['person'].value))
156 156          if self.who == "":
157 157              self.error = 'Your name is required. (blank)'
158 158      else: <$npage>
159          159      self.error = 'Your name is required. (missing)'
160          160 160
161 161      if form.has_key('cookie'):
162 162          self.cookies['user'] = unquote(strip(\
163 163              form['cookie'].value))
164 164      else: <$npage>
165          165      self.cookies['user'] = "
166          166 166
167 167      self.langs = []
168 168      if form.has_key('lang'):
169 169          langdata = form['lang']
170 170          if type(langdata) == type([]):
171 171              for eachLang in langdata:
172 172                  self.langs.append(eachLang.value)
173 173          else: <$npage>
174 174              self.langs.append(langdata.value)
175 175      else: <$npage>
176          176      self.error = 'At least one language required.'
177          177 177
178 178      if form.has_key('upfile'):
179 179          upfile = form["upfile"]
180 180          self.fn = upfile.filename or "
181 181          if upfile.file:
182 182              self.fp = upfile.file
183 183          else: <$npage>
184 184              self.fp = StringIO('(no data)')
185 185      else: <$npage>
186 186          self.fp = StringIO('(no file)')
187 187          self.fn = "
188 188
189 189      if not self.error:
190 190          self.doResults()
191 191      else: <$npage>
192          192      self.showError()
193          193 193
194 194      if __name__ == '__main__':
195 195          page = AdvCGI()
196 196          page.go()
197  <$npage>

```

advcgi.py looks strikingly similar to our friends3.py CGI scripts seen earlier in this chapter. It has form, results, and error pages to return. In addition to all of the advanced CGI features which are part of our new script, we are also using more of an object-oriented feel to our script by using a class with methods instead of just a set of functions. The HTML text for our pages are now static data for our class, meaning that they will remain constant across all instances—even though there is actually only one instance in our case.

Line-by-line (Block-by-block) explanation Lines 1 – 7

The usual start-up and import lines appear here. The only module you may not be familiar with is cStringIO, which we briefly introduced at the end of [Chapter 10](#) and also used in [Example 19-1](#). cStringIO.StringIO() creates a file-like object out of a string so that access to the string is similar to opening a file and using the handle to access the data.

Lines 9 – 12

After the AdvCGI class is declared, the header and url (static class) variables are created for use by the methods displaying all the different pages.

Lines 14 – 80

All the code in this block is used to generate and display the form page. The data attributes speak for themselves. getCPPCookies() obtains cookie information sent by the Web client, and showForm() collates all the information and sends the form page back to the client.

Lines 82 – 91

This block of code is responsible for the error page.

Lines 93 – 144

The results page is created using this block of code. The setCPPCookies() method requests that a client store the cookies for our application, and the doResults() method puts together all the data and sends the output back to the client.

Lines 146 – 196

The script begins by instantiating an AdvCGI page object, then call its go() method to start the ball rolling, in contrast to a strictly procedural programming process. The go() method contains the logic that reads all incoming data and decides which page to show.

The error page will be displayed if no name was given or if no languages were checked. The showForm() method is called to output the form if no input data was received, and the doResults() method is invoked otherwise to display the results page.

Handling the person field is the same as we have seen in the past, a single key-value pair; however, collecting the language information is a bit trickier since we must check for either a (Mini)FieldStorage instance or a list of such instances. We will employ the familiar type() built-in function for this purpose. In the end, we will have a list of a single language name or many, depending on the user's selections.

The use of cookies to contain data illustrates how they can be used to avoid using any kind of CGI field pass-through. You will notice in the code which obtains such data that no CGI processing is invoked, meaning that the data does not come from the FieldStorage object. The data is passed to us by the Web client with each request and the values (user's chosen data as well as information to fill in a succeeding form with pre-existing information) are obtained from cookies.

Because the showResults() method receives the new input from the user, it has the responsibility of setting the cookies, i.e., by calling setCPPCookies(). showForm() however, must read in the cookies' values in order to display a form page with the current user selections. This is done by its invocation of the getCPPCookies() method.

Finally, we get to the file upload processing. Regardless of whether a file was actually uploaded, FieldStorage is given a file handle in the file attribute. If the value attribute is accessed, then entire contents of the file will be placed into value. As a better alternative, you can access the file pointer—the file attribute—and perhaps read only one line at a time or other kind of slower processing.

In our case, file uploads are only part of user submissions, so we simply pass on the file pointer to the doResults() function to extract the data from the file. doResults() will display only the first 1K of the file for space reasons and to show you that it is not necessary (or necessarily productive/useful) to display a four megabyte binary file.

Discuss about Web (HTTP) Servers in detail.

Web (HTTP) Servers

Until now, we have been discussing the use of Python in creating Web clients and performing tasks to aid Web servers in CGI request processing. We know (and have seen earlier in [Sections 19.2](#) and [19.3](#)) that Python can be used to create both simple and complex Web clients. Complexity of CGI requests goes without saying.

However, we have yet to explore the creation of Web servers, and that is the focus of this section. If the Netscape, IE, Opera, Mozilla, and Lynx browsers are among the most popular Web clients, then what are the most common Web servers? They are Apache, Netscape, and IIS. In situations where these servers may be overkill for your desired application, we would like to use Python to help us create simple yet useful Web servers.

Creating Web Servers in Python

Since you have decided on building such an application, you will naturally be creating all the custom stuff, but all the base code you will need is already available in the Python Standard Library. To create a Web server, a base server and a "handler" are required.

The base (Web) server is a boilerplate item, a must have. Its role is to perform the necessary HTTP communication between client and server. The base server is (appropriately) named HTTPServer and is found in the BaseHTTPServer module.

The handler is the piece of software which does the majority of the "Web serving." It processes the client request and returns the appropriate file, whether static or

dynamically-generated by CGI. The complexity of the handler determines the complexity of your Web server. The Python standard library provides three different handlers.

The most basic, plain, vanilla handler, named BaseHTTPRequestHandler, is found in the BaseHTTPServer module, along with the base Web server. Other than taking a client request, no other handling is implemented at all, so you have to do it all yourself, such as in our myhttpd.pyserver below.

The SimpleHTTPRequestHandler, available in the SimpleHTTP-Server module, builds on BaseHTTPRequestHandler by implementing the standard GET and HEAD requests in a fairly straightforward manner. Still nothing sexy, but it gets the simple jobs done.

Finally, we have the CGIHTTPRequestHandler, available in the CGIHTTPServer module, which takes the SimpleHTTPRequestHandler and adds support for POST requests. It has the ability to call CGI scripts to perform the requested processing and can send the generated HTML back to the client.

The three modules and their classes are summarized in [Table 19-6](#).

Table 19.6. Web Server Modules and Classes

<i>Module</i>	<i>Description</i>
BaseHTTPServer	provides the base Web server and base handler classes, HTTPServer and BaseHTTPRequestHandler, respectively
SimpleHTTPServer	contains the SimpleHTTPRequestHandler class to perform GET and HEAD requests
CGIHTTPServer	contains the CGIHTTPRequestHandler class to process POST requests and perform CGI execution

To be able to understand how the more advanced handlers found in the SimpleHTTPServer and CGIHTTPServer modules work, we will implement simple GET processing for a BaseHTTPRequestHandler. In [Example 19-7](#), we present the code for a fully working Web server, myhttpd.py.

This server subclasses BaseHTTPRequestHandler and consists of a single do_GET() method, which is called when the base server receives a GET request. We attempt to open the path passed in by the client and if present, return an "OK" status (200) and forward the downloaded Web page. If the file was not found, returning a 404 status.

The main() function simply instantiates our Web server class and invokes it to run our familiar infinite server loop; shutting it down if interrupted by ^C or similar keystroke. If you have appropriate access and can run this server, you will notice that it displays loggable output which will look something like:

Example 19.7. Simple Web Server (myhttpd.py)

This simple Web server can read GET requests, fetch a Web page (.html file) and return it to the calling client. It uses the BaseHTTPRequestHandler found in BaseHTTPServer and implements the do_GET() method to enable processing of GET requests

```
<$nopcode>
001 1      #!/usr/bin/env python
002 2
003 3      from os import curdir, sep
004 4      from BaseHTTPServer import \
005 5          BaseHTTPRequestHandler, HTTPServer
006 6
007 7      class MyHandler(BaseHTTPRequestHandler):
008 8
009 9  def do_GET(self):
010 10          try: <$nopcode>
011 11              f = open(curdir + sep + self.path)
012 12              self.send_response(200)
013 13              self.send_header('Content-type',
014 14                  'text/html')
015 15              self.end_headers()
016 16              self.wfile.write(f.read())
017 17              f.close()
018 18          except IOError:
019 19              self.send_error(404, \
020 20                  20 'File Not Found: %s' % self.path)
021 21
022 22      def main():
023 23          try: <$nopcode>
024 24              24 server = HTTPServer(("", 80), MyHandler)
025 25              025 25      print 'Welcome to the machine...',
026 26              print 'Press ^C once or twice to quit.'
027 27              server.serve_forever()
028 28          except KeyboardInterrupt:
029 29              29 print '^C received, shutting down server'
030 30              030 30      server.socket.close()
031 31
032 32      if name_          == '_main_':
033 33          main()
034 <$nopcode>
```



```
# myhttpd.py
```

```
Welcome to the machine... Press ^C once or twice to quit localhost - - [26/Aug/2000 03:01:35] "GET /index.html HTTP/1.0" 200 -
```

```
localhost - - [26/Aug/2000 03:01:29] code 404, message File Not Found: /dummy.html
```

```
localhost - - [26/Aug/2000 03:01:29] "GET /dummy.html HTTP/1.0" 404 -
```

```
localhost - - [26/Aug/2000 03:02:03] "GET /hotlist.htm HTTP/1.0" 200 -
```

Of course, our simple little Web server is so simple, that it cannot even process plain text files. We leave that as an exercise for the reader, which can be found at the end of the chapter.

As you can see, it doesn't take much to have a Web server up and running in pure Python. There is plenty more you can do to enhance the handlers to customize it to your specific application. Please review the Library Reference for more information on these modules (and their classes) discussed in this section.

Related Modules

In [Table 19-7](#), we present a list of modules which you may find useful for Web and Internet development.

- The parsing modules deal with recognizing documents in specific formats.
- You can write POP- or IMAP-compliant mail clients using the corresponding protocol modules.
- Python has plenty of modules to support most kinds of binary file encoding for e-mail and other MIME-oriented applications.
- You can create clients for common Internet protocols like HTTP, FTP, Telnet, and NNTP with the appropriate modules. Be aware that urllib provides a high-level interface to protocols supported by your browser such as HTTP and FTP, so use of the lower-level protocol modules only makes sense when you cannot get all you want from urllib.
- Finally, we have the HTMLgen external module and the commercial Zope (Z Object Publishing Environment) system by Digital Creations. We introduced the HTMLgen module briefly at the end of [Section 19.5](#). It definitely comes in handy when you need to generate more complex HTML documents via CGI scripts.

Table 19.7. Web Programming Related Modules

<i>Module</i>	<i>Description</i>
Parsing	
htmllib	parses simple HTML files
sgmlib	parses simple SGML files
xmllib	parses simple XML files
robotparser ^[a]	parses robots.txt files for URL "fetchability" analysis
Mail Client Protocols	
poplib	use to create POP3 clients
imaplib	use to create IMAP4 clients
Mail and MIME Processing and Data Encoding Formats	
mailcap	parses mailcap files to obtain MIME application delegations
mimertools	provides functions for manipulating MIM-encoded messages
mimetypes	provides MIME type associations
MimeWriter	generates MIME-encoded multipart files
multifile	can parse multipart MIME-encoded files

quopri	en-/decodes data using quoted-printable encoding
rfc822	parses RFC822-compliant e-mail headers
smtplib	uses to create SMTP (Simple Mail Transfer Protocol) clients
base64	en-/decodes data using base64 encoding
binascii	en-/decodes data using base64, binhex, or uu (modules)
binhex	en-/decodes data using binhex4 encoding
uu	en-/decodes data using uuencode encoding
Internet Protocols	
httplib ^[a]	use to create HTTP (HyperText Transfer Protocol) clients (modified in Python 1.6 to support HTTP 1.1 and SSL)
ftplib	use to create FTP (File Transfer Protocol) clients
gopherlib	use to create Gopher clients
telnetlib	use to create Telnet clients
nntplib	use to create NNTP (Network News Transfer Protocol [Usenet]) clients
External/Commercial	
HTMLgen	use with CGI to generate complex HTML documents
Zope (<i>not a module</i>)	web object publishing product and Python Web application development environment (http://www.zope.org)

Zope is an open source Web publishing and application development platform which has Python code everywhere. Part of it is written in Python, and Python can be used to create extensions to Zope. Although it is in our Related Modules section, Zope is not a specific module as it is a powerful system for Web publishing.

Zope presents an extremely powerful alternative when simple CGI and database access just do not cut it for the application you are trying to build. Material on Zope itself can take up a book's length—you may even see one soon! We invite the reader to explore this system if desiring to create any complex system.

The robotparser module is new as of Python 1.6 and the httpplib and urllib modules have been modified for 1.6 to support HTTP connections over SSL.

MRITS

UNIT – V

Database Programming: Introduction, Python Database Application Programmer's Interface (DB-API), Object Relational Managers (ORMs), Related Modules

Explain about Database Programming.

Database Programming

Introduction

Persistent Storage

In any application, there is a need for persistent storage. Generally, there are three basic storage mechanisms: files, a relational database system (RDBMS), or some sort of hybrid, i.e., an API (application programmer interface) that "sits on top of" one of those existing systems, an object relational mapper (ORM), file manager, spreadsheet, configuration file, etc.

In an earlier chapter, we discussed persistent storage using both plain file access as well as a Python and DBM overlay on top of files, i.e., **dbm*, *dbhash/bsddb* files, *shelve* (combination of *pickle* and DBM), and using their dictionary-like object interface. This chapter will focus on using RDBMSs for the times when files or writing your own system does not suffice for larger projects.

Basic Database Operations and SQL

Before we dig into databases and how to use them with Python, we want to present a quick introduction (or review if you have some experience) to some elementary database concepts and the Structured Query Language (SQL).

Underlying Storage

Databases usually have a fundamental persistent storage using the file system, i.e., normal operating system files, special operating system files, and even raw disk partitions.

User Interface

Most database systems provide a command-line tool with which to issue SQL commands or queries. There are also some GUI tools that use the command-line clients or the database client library, giving users a much nicer interface.

Databases

An RDBMS can usually manage multiple databases, e.g., sales, marketing, customer support, etc., all on the same server (if the RDBMS is server-based; simpler systems are usually not). In the examples we will look at in this chapter, MySQL is an example of a server-based RDBMS because there is a server process running continuously waiting for commands while neither SQLite nor Gadgetfly have running servers.

Components

The *table* is the storage abstraction for databases. Each *row* of data will have fields that correspond to database *columns*. The set of table definitions of columns and data types per table all put together define the database *schema*.

Databases are *created* and *dropped*. The same is true for tables. Adding new rows to a database is called *inserting*, changing existing rows in a table is called *updating*, and removing existing rows in a table is called *deleting*. These actions are usually referred to as database *commands* or *operations*. Requesting rows from a database with optional criteria is called *querying*.

When you query a database, you can *fetch* all of the results (rows) at once, or just iterate slowly over each resulting row. Some databases use the concept of a *cursor* for issuing SQL commands, queries, and grabbing results, either all at once or one row at a time.

SQL

Database commands and queries are given to a database by SQL. Not all databases use SQL, but the majority of relational databases do. Here are some examples of SQL commands. Most databases are configured to be case-insensitive, especially database commands. The accepted style is to use CAPS for database keywords. Most command-line programs require a trailing semicolon (;) to terminate a SQL statement.

Creating a Database

```
CREATE DATABASE test;  
GRANT ALL ON test.* to user(s);
```

The first line creates a database named "test," and assuming that you are a database administrator, the second line can be used to grant permissions to specific users (or all of them) so that they can perform the database operations below.

Using a Database

```
USE test;
```

If you logged into a database system without choosing which database you want to use, this simple statement allows you to specify one with which to perform database operations.

Dropping a Database

```
DROP DATABASE test;
```

This simple statement removes all the tables and data from the database and deletes it from the system.

Creating a Table

```
CREATE TABLE users (login VARCHAR(8), uid INT, pridINT);
```

This statement creates a new table with a string column **login** and a pair of integer fields **uid** and **prid**.

Dropping a Table

```
DROP TABLE users;
```

This simple statement drops a database table along with all its data.

Inserting a Row

```
INSERT INTO users VALUES('leanna', 311, 1);
```

You can insert a new row in a database with the **INSERT** statement. Specify the table and the values that go into each field. For our example, the string 'leanna' goes into the **login** field, and **311** and **1** to **uid** and **prid**, respectively.

Updating a Row

```
UPDATE users SET prid=4 WHERE prid=2; UPDATE users SET prid=1 WHERE uid=311;
```

To change existing table rows, you use the **UPDATE** statement. Use **SET** for the columns that are changing and provide any criteria for determining which rows should change. In the first example, all users with a "project ID" or **prid** of 2 will be moved to project #4. In the second example, we take one user (with a **UID** of 311) and move them to project #1.

Deleting a Row

```
DELETE FROM users WHERE prid=%d; DELETE FROM users;
```

To delete a table row, use the **DELETE FROM** command, give the table you want to delete rows from, and any optional criteria. Without it, as in the second example, all rows will be deleted.

Now that you are up to speed on basic database concepts, it should make following the rest of the chapter and its examples much easier. If you need additional help, there are plenty of database books out in the market that you can check out.

Databases and Python

We are going to cover the Python database API and look at how to access relational databases from Python, either directly through a database interface, or via an ORM, and how you can accomplish the same task but without necessarily having to give explicitly commands in SQL.

Topics such as database principles, concurrency, schema, atomicity, integrity, recovery, proper complex left JOINS, triggers, query optimization, transactions, stored procedures, etc., are all outside the scope of this text, and we will not be discussing these in this chapter other than direct use from a Python application. There are plenty of resources you can refer to for general information. Rather, we will present how to store and retrieve data to/from RDBMSs while playing within a Python framework. You can then decide which is best for your current project or application and be able to study sample code that can get you started

instantly. The goal is to get you up to speed as quickly as possible if you need to integrate your Python application with some sort of database system.

We are also breaking out of our mode of covering only the "batteries included" features of the Python standard library. While our original goal was to play only in that arena, it has become clear that being able to work with databases is really a core component of everyday application development in the Python world.

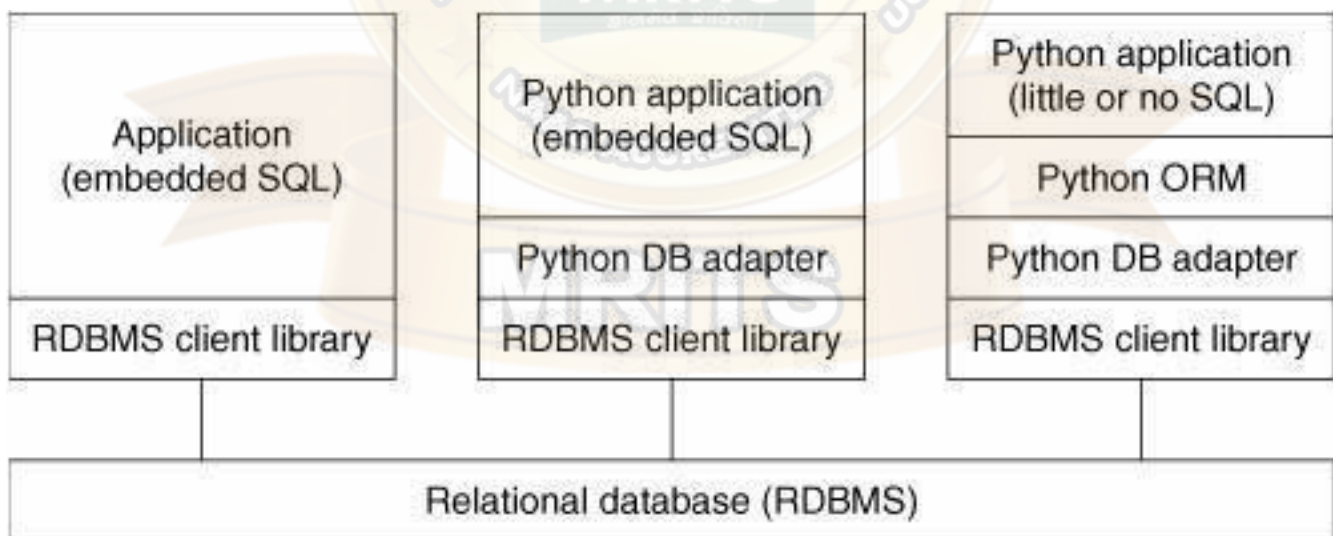
As a software engineer, you can probably only make it so far in your career without having to learn something about databases: how to use one (command-line and/or GUI interfaces), how to pull data out of one using the Structured Query Language (SQL), perhaps how to add or update information in a database, etc. If Python is your programming tool, then a lot of the hard work has already been done for you as you add database access to your Python universe. We first describe what the Python "DB-API" is, then give examples of database interfaces that conform to this standard.

We will give some examples using popular open source relational database management systems (RDBMSs). However, we will not include discussions of open source vs. commercial products, etc. Adapting to those other RDBMS systems should be fairly straightforward. A special mention will be given to Aaron Watters's Gadfly database, a simple RDBMS written completely in Python.

The way to access a database from Python is via an *adapter*. An adapter is basically a Python module that allows you to interface to a relational database's client library, usually in C. It is recommended that all Python adapters conform to the Python DB-SIG's Application Programmer Interface (API). This is the first major topic of this chapter.

[Figure 21.1](#) illustrates the layers involved in writing a Python database application, with and without an ORM. As you can see, the DB-API is your interface to the C libraries of the database client.

Figure 21-1. Multitiered communication between application and database. The first box is generally a C/C++ program while DB-API compliant adapters let you program applications in Python. ORMs can simplify an application by handling all of the database-specific details.



Write a note on Python Database Application Programmer's Interface (DB-API).

Python Database Application Programmer's Interface (DB-API)

Where can one find the interfaces necessary to talk to a database? Simple. Just go to the database topics section at the main Python Web site. There you will find links to the full and current DB-API (version 2.0), existing database modules, documentation, the special interest group, etc. Since its inception, the DB-API has been moved into PEP 249. (This PEP obsoletes the old DB-API 1.0 specification which is PEP 248.) What is the DB-API?

The API is a specification that states a set of required objects and database access mechanisms to provide

consistent access across the various database adapters and underlying database systems. Like most community-based efforts, the API was driven by strong need.

In the "old days," we had a scenario of many databases and many people implementing their own database adapters. It was a wheel that was being reinvented over and over again. These databases and adapters were implemented at different times by different people without any consistency of functionality. Unfortunately, this meant that application code using such interfaces also had to be customized to which database module they chose to use, and any changes to that interface also meant updates were needed in the application code.

A special interest group (SIG) for Python database connectivity was formed, and eventually, an API was born ... the DB-API version 1.0. The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler, usually only requiring tweaking several lines of code. You will see an example of this later on in this chapter.

Module Attributes

The DB-API specification mandates that the features and attributes listed below must be supplied. A DB-API-compliant module must define the global attributes as shown in [Table 21.1](#).

Table 21.1. DB-API Module Attributes

<i>Attribute</i>	<i>Description</i>
<code>apilevel</code>	Version of DB-API module is compliant with
<code>threadsafety</code>	Level of thread safety of this module
<code>paramstyle</code>	SQL statement parameter style of this module
<code>Connect()</code>	<code>Connect()</code> function
(Various exceptions)	(See Table 21.4)

Data Attributes

apilevel

This string (not float) indicates the highest version of the DB-API the module is compliant with, i.e., "1.0", "2.0", etc. If absent, "1.0" should be assumed as the default value.

threadsafety

This an integer with these possible values:

- 0: Not threadsafe, so threads should not share the module at all
- 1: Minimally threadsafe: threads can share the module but not connections
- 2: Moderately threadsafe: threads can share the module and connections but not cursors
- 3: Fully threadsafe: threads can share the module, connections, and cursors

If a resource is shared, a synchronization primitive such as a spin lock or semaphore is required for atomic-locking purposes. Disk files and global variables are not reliable for this purpose and may interfere with standard mutex operation. See the threading module or the chapter on multithreaded programming ([Chapter 16](#)) on how to use a lock.

paramstyle

The API supports a variety of ways to indicate how parameters should be integrated into an SQL statement that is eventually sent to the server for execution. This argument is just a string that specifies the form of string substitution you will use when building rows for a query or command (see [Table 21.2](#)).

Table 21.2. paramstyle Database Parameter Styles

<i>Parameter Style</i>	<i>Description</i>	<i>Example</i>
<code>numeric</code>	Numeric positional style	<code>WHERE name=:1</code>
<code>named</code>	Named style	<code>WHERE name=:name</code>
<code>pyformat</code>	Python dictionary <code>printf()</code> format conversion	<code>WHERE name=%(name)s</code>
<code>qmark</code>	Question mark style	<code>WHERE name=?</code>

format

ANSI C `printf()` format conversion

WHERE name=%s

Function Attribute(s)

`connect()` Function access to the database is made available through **Connection** objects. A compliant module has to implement a `connect()` function, which creates and returns a **Connection** object. [Table 21.3](#) shows the arguments to `connect()`.

Table 21.3. `connect()`

Function Attributes

<i>Parameter</i>	<i>Description</i>
<code>user</code>	Username
<code>password</code>	Password
<code>host</code>	Hostname
<code>database</code>	Database name
<code>dsn</code>	Data source name

You can pass in database connection information as a string with multiple parameters (DSN) or individual parameters passed as positional arguments (if you know the exact order), or more likely, keyworded arguments. Here is an example of using `connect()` from PEP 249:

```
connect(dsn='myhost:MYDB',user='guido',password='234$')
```

The use of DSN versus individual parameters is based primarily on the system you are connecting to. For example, if you are using an API like ODBC or JDBC, you would likely be using a DSN, whereas if you are working directly with a database, then you are more likely to issue separate login parameters. Another reason for this is that most database adapters have not implemented support for DSN. Below are some examples of non-DSN `connect()` calls. Note that not all adapters have implemented the specification exactly, e.g., `MySQLdb` uses `db` instead of `database`.

- `MySQLdb.connect(host='dbserv', db='inv', user='smith')`
- `PgSQL.connect(database='sales')`
- `psycopg.connect(database='template1', user='pgsql')`
- `gadfly.dbapi20.connect('csrDB', '/usr/local/database')`
- `sqlite3.connect('marketing/test')`

Exceptions

Exceptions that should also be included in the compliant module as globals are shown in [Table 21.4](#).

Table 21.4. DB-API Exception Classes

<i>Exception</i>	<i>Description</i>
<code>Warning</code>	Root warning exception class
<code>Error</code>	Root error exception class
<code>InterfaceError</code>	Database interface (not database) error
<code>DatabaseError</code>	Database error
<code>DataError</code>	Problems with the processed data
<code>OperationalError</code>	Error during database operation execution
<code>IntegrityError</code>	Database relational integrity error
<code>InternalError</code>	Error that occurs within the database
<code>ProgrammingError</code>	SQL command failed
<code>NotSupportedError</code>	Unsupported operation occurred

Connection Objects

Connections are how your application gets to talk to the database. They represent the fundamental communication mechanism by which commands are sent to the server and results returned. Once a connection has been established (or a pool of connections), you create cursors to send requests to and receive

replies from the database.

Methods

Connection objects are not required to have any data attributes but should define the methods shown in [Table 21.5](#).

Table 21.5. Connection Object Methods

<i>Method Name</i>	<i>Description</i>
<code>close()</code>	Close database connection
<code>commit()</code>	Commit current transaction
<code>rollback()</code>	Cancel current transaction
<code>cursor()</code>	Create (and return) a cursor or cursor-like object using this connection
<code>errorhandler(cxn, cur, errcls, errval)</code>	Serves as a handler for given connection cursor

When `close()` is used, the same connection cannot be used again without running into an exception. The `commit()` method is irrelevant if the database does not support transactions or if it has an auto-commit feature that has been enabled. You can implement separate methods to turn auto-commit off or on if you wish. Since this method is required as part of the API, databases that do not have the concept of transactions should just implement "pass" for this method.

Like `commit()`, `rollback()` only makes sense if transactions are supported in the database. After execution, `rollback()` should leave the database in the same state as it was when the transaction began. According to PEP 249, "Closing a connection without committing the changes first will cause an implicit rollback to be performed."

If the RDBMS does not support cursors, `cursor()` should still return an object that faithfully emulates or imitates a real cursor object. These are just the minimum requirements. Each individual adapter developer can always add special attributes specifically for their interface or database.

It is also recommended but not required for adapter writers to make all database module exceptions (see above) available via a connection. If not, then it is assumed that **Connection** objects will throw the corresponding module-level exception. Once you have completed using your connection and cursors closed, you should `commit()` any operations and `close()` your connection.

Cursor Objects

Once you have a connection, you can start talking to the database. As we mentioned above in the introductory section, a cursor lets a user issue database commands and retrieve rows resulting from queries. A Python DB-API cursor object functions as a cursor for you, even if cursors are not supported in the database. In this case, the database adapter creator must implement **CURSOR** objects so that they act like cursors. This keeps your Python code consistent when you switch between database systems that have or do not have cursor support.

Once you have created a cursor, you can execute a query or command (or multiple queries and commands) and retrieve one or more rows from the results set. [Table 21.6](#) shows data attributes and methods that cursor objects have.

Table 21.6. Cursor Object Attributes

Object Attribute	Description
<code>arraysize</code>	Number of rows to fetch at a time with <code>fetch many()</code> ; defaults to 1
<code>connection</code>	Connection that created this cursor (optional)
<code>description</code>	Returns cursor activity (7-item tuples): (name, type_code, display_size, internal_size, precision, scale, null_ok); only name and type_code are required
<code>lastrowid</code>	Row ID of last modified row (optional; if row IDs not supported, default to None)
<code>rowcount</code>	Number of rows that the last <code>execute*()</code> produced or affected
<code>callproc(func[, args])</code>	Call a stored procedure
<code>close()</code>	Close cursor
<code>execute(op[, args])</code>	Execute a database query or command
<code>executemany(op, args)</code>	Like <code>execute()</code> and <code>map()</code> combined; prepare and execute a database query or command over given arguments
<code>fetchone()</code>	Fetch next row of query result
<code>fetchmany([size=cursor.arraysize])</code>	Fetch next size rows of query result
<code>fetchall()</code>	Fetch all (remaining) rows of a query result
<code>__iter__()</code>	Create iterator object from this cursor (optional; also see <code>next()</code>)
<code>messages</code>	List of messages (set of tuples) received from the database for cursor execution (optional)
<code>next()</code>	Used by iterator to fetch next row of query result (optional; like <code>fetchone()</code> , also see <code>__iter__()</code>)
<code>nextset()</code>	Move to next results set (if supported)
<code>rownumber</code>	Index of cursor (by row, 0-based) in current result set (optional)
<code>setinput-sizes(sizes)</code>	Set maximum input-size allowed (required but implementation optional)
<code>setoutput size(size[, col])</code>	Set maximum buffer size for large column fetches (required but implementation optional)

The most critical attributes of cursor objects are the `execute*()` and the `fetch*()` methods ... all the service requests to the database are performed by these. The `arraysize` data attribute is useful in setting a default size for `fetchmany()`. Of course, closing the cursor is a good thing, and if your database supports stored procedures, then you will be using `callproc()`.

Type Objects and Constructors

Oftentimes, the interface between two different systems is the most fragile. This is seen when converting Python objects to C types and vice versa. Similarly, there is also a fine line between Python objects and native database objects. As a programmer writing to Python's DB-API, the parameters you send to a database are given as strings, but the database may need to convert it to a variety of different, supported data types that are correct for any particular query.

For example, should the Python string be converted to a VARCHAR, a TEXT, a BLOB, or a raw BINARY object, or perhaps a DATE or TIME object if that is what the string is supposed to be? Care must be taken to provide database input in the expected format, so because of this another requirement of the DB-API is to create constructors that build special objects that can easily be converted to the appropriate database objects. [Table 21.7](#) describes classes that can be used for this purpose. SQL NULL

values are mapped to and from Python's NULL object, `None`.

Table 21.7. Type Objects and Constructors

<i>Type Object</i>	<i>Description</i>
<code>Date(yr, mo, dy)</code>	Object for a date value
<code>Time(hr, min, sec)</code>	Object for a time value
<code>Timestamp(yr, mo, dy, hr, min, sec)</code>	Object for a timestamp value
<code>DateFromTicks(ticks)</code>	Date object given number of seconds since the epoch
<code>TimeFromTicks(ticks)</code>	Time object given number of seconds since the epoch
<code>TimestampFromTicks(ticks)</code>	Timestamp object given number of seconds since the epoch
<code>Binary(string)</code>	Object for a binary (long) string value
<code>STRING</code>	Object describing string-based columns, e.g., VARCHAR
<code>BINARY</code>	Object describing (long) binary columns, i.e., RAW, BLOB
<code>NUMBER</code>	Object describing numeric columns
<code>DATETIME</code>	Object describing date/time columns
<code>ROWID</code>	Object describing "row ID" columns

Changes to API Between Versions

Several important changes were made when the DB-API was revised from version 1.0 (1996) to 2.0 (1999):

- Required `dbi` module removed from API
- Type objects were updated
- New attributes added to provide better database bindings
- `callproc()` semantics and return value of `execute()` redefined
- Conversion to class-based exceptions

Since version 2.0 was published, some of the additional optional DB-API extensions that you read about above were added in 2002. There have been no other significant changes to the API since it was published. Continuing discussions of the API occur on the DB-SIG mailing list. Among the topics brought up over the last 5 years include the possibilities for the next version of the DB-API, tentatively named DB-API 3.0. These include the following:

- Better return value for `nextset()` when there is a new result set
- Switch from `float` to `Decimal`
- Improved flexibility and support for parameter styles
- Prepared statements or statement caching
- Refine the transaction model
- State the role of API with respect to portability
- Add unit testing

Relational Databases

So, you are now ready to go. A burning question must be, "Interfaces to which database systems are available to me in Python?" That inquiry is similar to, "Which platforms is Python available for?" The answer is, "Pretty much all of them." Following is a list that is comprehensive but not exhaustive:

Commercial RDBMSs

- Informix
- Sybase
- Oracle
- MS SQL Server
- DB/2
- SAP
- Interbase
- Ingres

Open Source RDBMSs

- MySQL
- PostgreSQL
- SQLite

- Gadfly
- Database APIs**
- JDBC
 - ODBC

Databases and Python: Adapters

For each of the databases supported, there exists one or more adapters that let you connect to the target database system from Python. Some databases, such as Sybase, SAP, Oracle, and SQLServer, have more than one adapter available. The best thing to do is to find out which ones fit your needs best. Your questions for each candidate may include: how good its performance is, how useful is its documentation and/or Web site, whether it has an active community or not, what the overall quality and stability of the driver is, etc. You have to keep in mind that most adapters provide just the basic necessities to get you connected to the database. It is the extras that you may be looking for. Keep in mind that you are responsible for higher-level code like threading and thread management as well as management of database connection pools, etc.

Let us now look at some examples of how to use an adapter module to talk to a relational database. The real secret is in setting up the connection. Once you have this and use the DB-API objects, attributes, and object methods, your core code should be pretty much the same regardless of which adapter and RDBMS you use.

Examples of Using Database Adapters

First, let us look at a some sample code, from creating a database to creating a table and using it. We present examples using MySQL, PostgreSQL, and SQLite.

MySQL

We will use MySQL as the example here, along with the only MySQL Python adapter: **MySQLdb**, aka MySQL-python. In the various bits of code, we will also show you (deliberately) examples of error situations so that you have an idea of what to expect, and what you may wish to create handlers for.

We first log in as an administrator to create a database and grant permissions, then log back in as a normal client.

```
>>> import MySQLdb
>>> cxn = MySQLdb.connect(user='root')
>>> cxn.query('DROP DATABASEtest') Traceback (most recent calllast):
File "<stdin>", line 1, in ?
_mysql_exceptions.OperationalError:(1008,"Can'tdrop database 'test'; database doesn't exist")
>>> cxn.query('CREATE DATABASE test')
>>> cxn.query("GRANT ALL ON test.* to '@'localhost'")
>>> cxn.commit()
>>> cxn.close()
```

In the code above, we did not use a cursor. Some adapters have **Connection**objects, which can execute SQL queries with the **query()** method, but not all. We recommend you either not use it or check your adapter to make sure it is available.

The **commit()**was optional for us as auto-commit is turned on by default in MySQL. We then connect back to the new database as a regular user, create a table, and perform the usual queries and commands using SQL to get our job done via Python. This time we use cursors and their **execute()** method.

The next set of interactions shows us creating a table. An attempt to create it again (without first dropping it) results in an error.

```
>>> cxn = MySQLdb.connect(db='test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), uid INT)') 0L
```

Now we will insert a few rows into the database and query them out.

```
>>> cur.execute("INSERT INTO users VALUES('john', 7000)") 1L
>>> cur.execute("INSERT INTO users VALUES('jane', 7001)") 1L
```



```

>>> cur.execute("INSERT INTO users VALUES('bob', 7200)") 1L
>>> cur.execute("SELECT * FROM users WHERE login LIKE 'j%'") 2L
>>> for data in cur.fetchall():
...     print '%s\t%s' % data
...
john          7000
jane          7001

```

The last bit features updating the table, either updating or deleting rows.

```

>>> cur.execute("UPDATE users SET uid=7100 WHERE uid=7001") 1L
>>> cur.execute("SELECT * FROM users") 3L
>>> for data in cur.fetchall():
...     print '%s\t%s' % data
...
john          7000
jane          7100
bob           7200
>>> cur.execute("DELETE FROM users WHERE login='bob'") 1L
>>> cur.execute("DROP TABLE users") 0L
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()

```

MySQL is one of the most popular open source databases in the world, and it is no surprise that a Python adapter is available for it. Keep in mind that no database modules are available in the Python standard library; all adapters are third-party packages that have to be downloaded and installed separately from Python. Please see the References section toward the end of the chapter to find out how to download it.

PostgreSQL

Another popular open source database is PostgreSQL. Unlike MySQL, there are no less than three current Python adapters available for Postgres: **psycopg**, **PyPgSQL**, and **PyGreSQL**. A fourth, **PoPy**, is now defunct, having contributed its project to combine with that of **PyGreSQL** back in 2003. Each of the three remaining adapters has its own characteristics, strengths, and weaknesses, so it would be a good idea to practice due diligence to determine which is right for you.

The good news is that the interfaces are similar enough that you can create an application that, say, measures the performance between all three (if that is a metric that is important to you). Here we show you the setup code to get a **Connection** object for each:

psycopg

```

>>> import psycopg
>>> cxn = psycopg.connect(user='pgsql')

```

PyPgSQL

```

>>> from pyPgSQL import PgSQL
>>> cxn = PgSQL.connect(user='pgsql')

```

PyGreSQL

```

>>> import pgdb
>>> cxn = pgdb.connect(user='pgsql')

```

Now comes some generic code that will work for all three adapters.

```

>>> cur = cxn.cursor()
>>> cur.execute('SELECT * FROM pg_database')
>>> rows = cur.fetchall()

```



```
>>> for i in rows:
...     print i
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

Finally, you can see how their outputs are slightly different from one another.

PyPgSQL

```
sales template1 template0
```

psycopg

```
('sales', 1, 0, 0, 1, 17140, '140626', '3221366099',
", None, None)
('template1', 1, 0, 1, 1, 17140, '462', '462', ",None,
'{pgsql=C*T*/pgsql}')
('template0', 1, 0, 1, 0, 17140, '462', '462', ",None,
'{pgsql=C*T*/pgsql}')
```

PyGreSQL

```
['sales', 1, 0, False, True, 17140L, '140626', '3221366099', ", None, None]
['template1', 1, 0, True, True, 17140L, '462', '462',", None, '{pgsql=C*T*/pgsql}']
['template0', 1, 0, True, False, 17140L, '462', '462', ", None, '{pgsql=C*T*/pgsql}']
```

SQLite

For extremely simple applications, using files for persistent storage usually suffices, but the most complex and data-driven applications demand a full relational database. SQLite targets the intermediate systems and indeed is a hybrid of the two. It is extremely lightweight and fast, plus it is serverless and requires little or no administration.

SQLite has seen a rapid growth in popularity, and it is available on many platforms. With the introduction of the `pysqlite` database adapter in Python 2.5 as the `sqlite3` module, this marks the first time that the Python standard library has featured a database adapter in any release.

It was bundled with Python not because it was favored over other databases and adapters, but because it is simple, uses files (or memory) as its backend store like the DBM modules do, does not require a server, and does not have licensing issues. It is simply an alternative to other similar persistent storage solutions included with Python but which happens to have a SQL interface.

Having a module like this in the standard library allows users to develop rapidly in Python using SQLite, then migrate to a more powerful RDBMS such as MySQL, PostgreSQL, Oracle, or SQL Server for production purposes if this is their intention.

Although the database adapter is now provided in the standard library, you still have to download the actual database software yourself. However, once you have installed it, all you need to do is start up Python (and import the adapter) to gain immediate access:

```
>>> import sqlite3
>>> cxn = sqlite3.connect('sqlite_test/test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), uid INTEGER)')
>>> cur.execute('INSERT INTO users VALUES("john", 100)')
>>> cur.execute('INSERT INTO users VALUES("jane", 110)')
>>> cur.execute('SELECT * FROM users')
>>> for eachUser in cur.fetchall():
```

```

...         print eachUser
...
(u'john', 100)
(u'jane', 110)
>>> cur.execute('DROP TABLE users')
<sqlite3.Cursor object at 0x3d4320>
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()

```

Okay, enough of the small examples. Next, we look at an application similar to our earlier example with MySQL, but which does a few more things:

- Creates a database (if necessary)
- Creates a table
- Inserts rows into the table
- Updates rows in the table
- Deletes rows from the table
- Drops the table

For this example, we will use two other open source databases. SQLite has become quite popular of late. It is very small, lightweight, and extremely fast for all the most common database functions. Another database involved in this example is Gadfly, a mostly SQL-compliant RDBMS written entirely in Python. (Some of the key data structures have a C module available, but Gadfly can run without it [slower, of course].)

Some notes before we get to the code. Both SQLite and Gadfly require the user to give the location to store database files (while MySQL has a default area and does not require this information from the user). The most current incarnation of Gadfly is not yet fully DB-API 2.0 compliant, and as a result, is missing some functionality, most notably the cursor attribute `rowcount` in our example.

Database Adapter Example Application

In the example below, we want to demonstrate how to use Python to access a database. In fact, for variety, we added support for three different database systems: Gadfly, SQLite, and MySQL. We are going to create a database (if one does not already exist), then run through various database operations such as creating and dropping tables, and inserting, updating, and deleting rows. [Example 21.1](#) will be duplicated for the upcoming section on ORMs as well.

Example 21.1. Database Adapter Example (`ushuffle_db.py`)

This script performs some basic operations using a variety of databases (MySQL, SQLite, Gadfly) and a corresponding Python database adapter.

```

1      #!/usr/bin/envpython 2
3      import os
4      from random import randrange as rrange 5
6      COLSIZ = 10
7      RDBMSs = {'s': 'sqlite', 'm': 'mysql', 'g': 'gadfly'}
8      DB_EXC = None 9
10     def setup():
11         return RDBMSs[raw_input("
12     Choose a database system:
13
14     (M)ySQL
15     (G)adfly
16     (S)QLite 17
18     Enterchoice:").strip().lower()[0]] 19
20     def connect(db, dbName):

```

```

21  global DB_EXC
22  dbDir = '%s_%s' % (db, dbName)
23
24  if db == 'sqlite':
25      try:
26          import sqlite3
27      except ImportError, e:
28          try:
29              from pysqlite2 import dbapi2 as sqlite3
30          except ImportError, e:
31              return None
32
33      DB_EXC = sqlite3
34      if not os.path.isdir(dbDir):
35          os.mkdir(dbDir)
36      cxn=sqlite.connect(os.path.join(dbDir,dbName)) 37
38  elif db == 'mysql':
39      try:
40          import MySQLdb
41          import _mysql_exceptions as DB_EXC
42      except ImportError, e:
43          return None
44
45      try:
46          cxn = MySQLdb.connect(db=dbName)
47      except _mysql_exceptions.OperationalError, e:
48          cxn = MySQLdb.connect(user='root')
49      try:
50          cxn.query('DROP DATABASE %s' % dbName)
51      except DB_EXC.OperationalError, e:
52          pass
53          cxn.query('CREATE DATABASE %s' % dbName)
54          cxn.query("GRANT ALL ON %s.* to '@localhost'" % dbName)
55          cxn.commit()
56          cxn.close()
57          cxn =MySQLdb.connect(db=dbName) 58
59  elif db == 'gadfly':
60      try:
61          from gadfly import gadfly
62          DB_EXC = gadfly
63      except ImportError, e:
64          return None
65
66      try:
67          cxn = gadfly(dbName, dbDir)
68      except IOError, e:
69          cxn = gadfly()
70          if not os.path.isdir(dbDir):
71              os.mkdir(dbDir)
72          cxn.startup(dbName, dbDir)
73
74      else:
75          return None
76      return cxn
77  def create(cur):
78      try
79          cur.execute("""
80          CREATE TABLE users (

```

```

81         login VARCHAR(8),
82         uid INTEGER,
83         prid INTEGER) 84 ""
85     except DB_EXC.OperationalError, e:
86         drop(cur)
87         create(cur) 88
89     drop = lambda cur: cur.execute('DROP TABLE users') 90
91     NAMES = (
92         ('aaron', 8312), ('angela', 7603), ('dave', 7306),
93         ('davina',7902), ('elliott', 7911), ('ernie', 7410),
94         ('jess', 7912), ('jim', 7512), ('larry', 7311),
95         ('leslie', 7808), ('melissa', 8602), ('pat', 7711),
96         ('serena', 7003), ('stan', 7607), ('faye', 6812),
97         ('amy', 7209),
98     )
99
100    def randName():
101        pick = list(NAMES)
102        while len(pick) > 0:
103            yield pick.pop(rrange(len(pick))) 104
104    def insert(cur, db):
105        if db == 'sqlite':
106            cur.executemany("INSERT INTO users VALUES(?, ?, ?)",
107                [(who, uid, rrange(1,5)) for who, uid in inrandName()])
108        elif db == 'gadfly':
109            for who, uid in randName():
110                cur.execute("INSERT INTO users VALUES(?, ?, ?)",
111                    (who, uid, rrange(1,5)))
112        elif db == 'mysql':
113            cur.executemany("INSERT INTO users VALUES(%s, %s, %s)",
114                [(who, uid, rrange(1,5)) for who, uid in randName()]) 116
115    getRC = lambda cur: cur.rowcount if hasattr(cur, 'rowcount') else -1
116
117
118
119    def update(cur):
120        fr =rrange(1,5)
121        to =rrange(1,5)
122        cur.execute(
123            "UPDATE users SET prid=%d WHERE prid=%d" % (to,fr))
124        return fr, to,getRC(cur) 125
125    def delete(cur):
126        rm = range(1,5)
127        cur.execute('DELETE FROM users WHERE prid=%d' % rm)
128        return rm,getRC(cur) 130
129
130    def dbDump(cur):
131        cur.execute('SELECT * FROM users')
132        print "\n%s%s%s" % ('LOGIN'.ljust(COLSIZ),
133            'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
134        for data in cur.fetchall():
135            print '%s%s%s' % tuple([str(s).title().ljust(COLSIZ) \
136                for s in data]) 138
137
138    def main():
139        db = setup()
140        print '*** Connecting to %r database' % db
141        cxn = connect(db, 'test')
142        if not cxn:
143            print 'ERROR: %r not supported, exiting' % db
144        return
145

```



```

146     cur =cxn.cursor() 147
148     print '\n*** Creating users table'
149         create(cur) 150
151     print '\n*** Inserting names into table'
152     insert(cur, db)
153     dbDump(cur) 154
155     print '\n*** Randomly moving folks',
156     fr, to, num = update(cur)
157     print 'from one group (%d) to another (%d)' % (fr,to)
158     print '\t(%d users moved)' % num
159     dbDump(cur) 160
161     print '\n*** Randomly choosing group',
162     rm, num = delete(cur)
163     print '(%d) to delete' % rm
164     print '\t(%d users removed)' % num
165     dbDump(cur) 166
167     print '\n*** Dropping users table'
168     drop(cur)
169     cur.close()
170     cxn.commit()
171     cxn.close() 172
173     if name _ == '_main_':
174         main()

```

Line-by-Line Explanation Lines 118

The first part of this script imports the necessary modules, creates some global "constants" (the column size for display and the set of databases we are supporting), and features the `setup()` function, which prompts the user to select the RDBMS to use for any particular execution of this script.

The most notable constant here is `DB_EXC`, which stands for DataBase EXception. This variable will eventually be assigned the database exception module for the specific database system that the users chooses to use to run this application with. In other words, if users choose MySQL, `DB_EXC` will be `_mysql_exceptions`, etc. If we developed this application in more of an object-oriented fashion, this would simply be an instance attribute, i.e., `self.db_exc_module` or something like that.

Lines 2075

The guts of consistent database access happens here in the `connect()` function. At the beginning of each section, we attempt to load the requested database modules. If a suitable one is not found, `None` is returned to indicate that the database system is not supported.

Once a connection is made, then all other code is database and adapter independent and should work across all connections. (The only exception in our script is `insert()`.) In all three subsections of this set of code, you will notice that a valid connection should be passed back as `cxn`.

If SQLite is chosen (lines 24-36), we attempt to load a database adapter. We first try to load the standard library's `sqlite3` module (Python 2.5+). If that fails, we look for the third-party `pysqlite2` package. This is to support 2.4.x and older systems with the `pysqlite` adapter installed. If a suitable adapter is found, we then check to ensure that the directory exists because the database is file based. (You may also choose to create an in-memory database.) When the `connect()` call is made to SQLite, it will either use one that already exists or make a new one using that path if it does not.

MySQL (lines 38-57) uses a default area for its database files and does not require this to come from the user. Our code attempts to connect to the specified database. If an error occurs, it could mean either that the database does not exist or that it does exist but we do not have permission to see it. Since this is just a test application, we elect to drop the database altogether (ignoring any error if the database does not exist), and re-create it, granting all permissions after that.

The last database supported by our application is Gadfly (lines 59-75). (At the time of writing, this database is mostly but not fully DB-API-compliant, and you will see this in this application.) It uses a startup mechanism similar to that of SQLite: it starts up with the directory where the database files should be. If it is there, fine, but if not, you have to take a roundabout way to start up a new database. (Why this is, we are not sure. We believe that the `startup()` functionality should be merged into that of the constructor `gadfly.gadfly()`.)

Lines 7789

The `create()` function creates a new users table in our database. If there is an error, that is almost always because the table already exists. If this is the case, drop the table and re-create it by recursively calling this function again. This code is dangerous in that if the recreation of the table still fails, you will have infinite recursion until your application runs out of memory. You will fix this problem in one of the exercises at the end of the chapter.

The table is dropped from the database with the one-liner `drop()`.

Lines 91103

This is probably the most interesting part of the code outside of database activity. It consists of a constant set of names and user IDs followed by the generator `randName()` whose code can be found in [Chapter 11 \(Functions\)](#) in [Section 11.10](#). The `NAMES` constant is a tuple that must be converted to a list for use with `randName()` because we alter it in the generator, randomly removing one name at a time until the list is exhausted. Well, if `NAMES` was a list, we would only use it once. Instead, we make it a tuple and copy it to a list to be destroyed each time the generator is used.

Lines 105115

The `insert()` function is the only other place where database-dependent code lives, and the reason is that each database is slightly different in one way or another. For example, both the adapters for SQLite and MySQL are DB-API-compliant, so both of their cursor objects have an `executemany()` function, whereas Gadfly does not, so rows have to be inserted one at a time.

Another quirk is that both SQLite and Gadfly use the `qmark` parameter style while MySQL uses `format`. Because of this, the format strings are different. If you look carefully, however, you will see that the arguments themselves are created in a very similar fashion.

What the code does is this: for each name-userID pair, it assigns that individual to a project group (given by its project ID or `prid`). The project ID is chosen randomly out of four different groups (`randrange(1,5)`).

Line 117

This single line represents a conditional expression (read as: Python ternary operator) that returns the rowcount of the last operation (in terms of rows altered), or if the cursor object does not support this attribute (meaning it is not DB-API-compliant), it returns -1.

Conditional expressions were added in Python 2.5, so if you are using 2.4.x or older, you will need to convert it back to the "old-style" way of doing it:

```
getRC = lambda cur: (hasattr(cur, 'rowcount') \
and [cur.rowcount] or [-1])[0]
```

If you are confused by this line of code, don't worry about it. Check the FAQ to see why this is, and get a taste of why conditional expressions were finally added to Python in 2.5. If you *are* able to figure it out, then you have developed a solid understanding of Python objects and their Boolean values.

Lines 119129

The `update()` and `delete()` functions randomly choose folks from one group. If the operation is update, move them from their current group to another (also randomly chosen); if it is delete, remove them altogether.

Lines 131137

The `dbDump()` function pulls all rows from the database, formats them for printing, and displays them to the user. The `print` statement to display each user is the most obfuscated, so let us take it apart.

First, you should see that the data were extracted after the `SELECT` by the `fetchall()` method. So as we iterate each user, take the three columns (`login`, `uid`, `prid`), convert them to strings (if they are not already), titlecase it, and format the complete string to be `COLSIZ` columns left-justified (right-hand space padding). Since the code to generate these three strings is a list (via the list comprehension), we need to convert it to a tuple for the format operator (`%`).

Lines 139174

The director of this movie is `main()`. It makes the individual functions to each function described above that defines how this script works (assuming that it does not exit due to either not finding a database adapter or not being able to obtain a connection [lines 143-145]). The bulk of it should be fairly self-explanatory given the proximity of the `print` statements. The last bits of `main()` close the cursor, and commit and close the connection. The final lines of the script are the usual to start the script.

Discuss in detail about Object-Relational Managers (ORMs).

Object-Relational Managers (ORMs)

As seen in the previous section, a variety of different database systems are available today, and most of them have Python interfaces to allow you to harness their power. The only drawback to those systems is the need to know SQL. If you are a programmer who feels more comfortable with manipulating Python objects instead of SQL queries, yet still want to use a relational database as your data backend, then you are a great candidate to be a user of ORMs.

Think Objects, Not SQL

Creators of these systems have abstracted away much of the pure SQL layer and implemented objects in Python that you can manipulate to accomplish the same tasks without having to generate the required lines of SQL. Some systems allow for more flexibility if you do have to slip in a few lines of SQL, but for the most part, you can avoid almost all the general SQL required.

Database tables are magically converted to Python classes with columns and features as attributes and methods responsible for database operations. Setting up your application to an ORM is somewhat similar to that of a standard database adapter. Because of the amount of work that ORMs perform on your behalf, some things are actually more complex or require more lines of code than using an adapter directly. Hopefully, the gains you achieve in productivity make up for a little bit of extra work.

Python and ORMs

The most well-known Python ORMs today are SQLAlchemy and SQLAlchemy. We will give you examples of SQLAlchemy and SQLAlchemy because the systems are somewhat disparate due to different philosophies, but once you figure these out, moving on to other ORMs is much simpler.

Some other Python ORMs include PyDO/PyDO2, PDO, Dejavu, PDO, Durus, QLine, and ForgetSQL. Larger Web-based systems can also have their own ORM component, i.e., WebWare MiddleKit and Django's Database API. Note that "well-known" does not mean "best for your application." Although these others were not included in our discussion, that does not mean that they would not be right for your application.

Employee Role Database Example

We will port our user shuffle application `ushuffle_db.py` to both SQLAlchemy and SQLAlchemy below. MySQL will be the backend database server for both. You will note that we implement these as classes because there is more of an object "feel" to using ORMs as opposed to using raw SQL in a database adapter. Both examples import the set of `NAMES` and the random name chooser from `ushuffle_db.py`.

This is to avoid copying-and-pasting the same code everywhere as code reuse is a good thing.

SQLAlchemy

We start with SQLAlchemy because its interface is somewhat closer to SQL than SQLAlchemy's interface. SQLAlchemy abstracts really well to the object world but does give you more flexibility in issuing SQL if you have to. You will find both of these ORMs ([Examples 21.2](#) and [21.3](#)) very similar in terms of setup and access, as well as being of similar size, and both shorter than `ushuffle_db.py` (including the sharing of the names list and generator used to randomly iterate through that list).

Example 21.2. SQLAlchemy ORM Example (`ushuffle_sa.py`)

This "user shuffle" application features SQLAlchemy paired up with the MySQL database as its backend.

```
1      #!/usr/bin/envpython 2
3      import os
4      from random import randrange as rrange
5      from sqlalchemy import *
6      from ushuffle_db import NAMES, randName 7
7      FIELDS = ('login', 'uid', 'prid')
8      DBNAME = 'test'
9      COLSIZ = 10 11
10     class MySQLAlchemy(object):
11         def init (self, db, dbName):
12             import MySQLdb
13             import _mysql_exceptions
14             MySQLdb = pool.manage(MySQLdb)
15             url = 'mysql://db=%s' % DBNAME
16             eng = create_engine(url)
17             try:
18                 cxn = eng.connection()
19             except _mysql_exceptions.OperationalError, e:
20                 eng1 = create_engine('mysql://user=root')
21                 try:
22                     eng1.execute('DROP DATABASE %s' % DBNAME)
23                 except _mysql_exceptions.OperationalError, e:
24                     pass
25                 eng1.execute('CREATE DATABASE %s' % DBNAME)
26                 eng1.execute(
27                     "GRANT ALL ON %s.* TO '@localhost'" % DBNAME)
28                 eng1.commit()
29                 cxn =eng.connection() 32
30             try:
31                 users = Table('users', eng, autoload=True)
32             except exceptions.SQLError, e:
33                 users = Table('users', eng,
34                     Column('login', String(8)),
35                     Column('uid', Integer),
36                     Column('prid', Integer),
37                     redefine=True)
38
39             self.eng =eng
40             self.cxn =cxn
41             self.users =users 45
42         def create(self):
43             users = self.users
44             try:
45                 users.drop()
46             except exceptions.SQLError, e:
47                 pass
48
49
50
51
```



```

52         users.create()
53
54     def insert(self):
55         d = [dict(zip(FIELDS,
56                     [who, uid, rrange(1,5)])) for who, uid in randName()]
57         return self.users.insert().execute(*d).rowcount
58
59     def update(self):
60         users = self.users
61         fr = rrange(1,5)
62         to = rrange(1,5)
63         return fr, to, \ users.update(users.c.prid==fr).execute(prid=to).rowcount
64
65
66     def delete(self):
67         users = self.users
68         rm = rrange(1,5)
69         return rm, \ users.delete(users.c.prid==rm).execute().rowcount
70
71
72     def dbDump(self):
73         res = self.users.select().execute()
74         print '\n%s%s%s' % ('LOGIN'.ljust(COLSIZ),
75                             'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
76         for data in res.fetchall():
77             print'%s%s%s'%tuple([str(s).title().ljust (COLSIZ) for s in data])
78
79     def getattr_(self, attr):
80         return getattr(self.users, attr)
81
82     def finish(self):
83         self.cxn.commit()
84         self.eng.commit()
85
86     def main():
87         print '*** Connecting to %r database' %DBNAME
88         orm = MySQLAlchemy('mysql', DBNAME)
89
90         print '\n*** Creating users table'
91         orm.create()
92
93         print '\n*** Inserting names intotable'
94         orm.insert()
95         orm.dbDump()
96
97         print '\n*** Randomly moving folks',
98         fr, to, num = orm.update()
99         print 'from one group (%d) to another (%d)' % (fr, to)
100        print '\t(%d users moved)' % num
101        orm.dbDump()
102
103        print '\n*** Randomly choosing group',
104        rm, num = orm.delete()
105        print '(%d) to delete' % rm
106        print '\t(%d users removed)' % num
107        orm.dbDump()

```

```

108
109 print '\n*** Dropping users table'
110 orm.drop()
111 orm.finish()
112
113 if name == ' main ':
114     main()

```

Example 21.3. SQLAlchemy ORM Example (`ushuffle_so.py`)

This "user shuffle" application features SQLAlchemy paired up with the MySQL database as its backend.

```

1  #!/usr/bin/env python
2
3  import os
4  from random import randrange as rrange
5  from sqlalchemy import *
6  from ushuffle_db import NAMES, randName
7
8  DBNAME = 'test'
9  COLSIZ = 10
10  FIELDS = ('login', 'uid', 'prid')
11
12  class MySQLObject(object):
13  def init(self, db, dbName):
14  import MySQLdb
15  import _mysql_exceptions
16  url = 'mysql://localhost/%s' % DBNAME
17
18  while True:
19  cxn = connectionForURI(url)
20  sqlhub.processConnection = cxn
21  #cxn.debug = True
22  try:
23  class Users(SQLObject):
24  class sqlmeta:
25  fromDatabase = True
26  login = StringCol(length=8)
27  uid = IntCol()
28  prid = IntCol()
29  break
30  except _mysql_exceptions.ProgrammingError, e:
31  class Users(SQLObject):
32  login = StringCol(length=8)
33  uid = IntCol()
34  prid = IntCol()
35  break
36  except _mysql_exceptions.OperationalError, e:
37  cxn1 = sqlhub.processConnection=
connectionForURI('mysql://root@localhost')
38  cxn1.query("CREATE DATABASE %s" % DBNAME)

```

```

39         cxn1.query("GRANT ALL ON %s.* TO '@' localhost" % DBNAME)
40         cxn1.close()
41         self.users = Users
42         self.cxn =cxn 43
44     def create(self):
45         Users = self.users
46         Users.dropTable(True)
47         Users.createTable() 48
49     def insert(self):
50         for who, uid in randName():
51             self.users(**dict(zip(FIELDS,
52                 [who, uid,rrange(1,5)]))) 53
54     def update(self):
55         fr =rrange(1,5)
56         to =rrange(1,5)
57         users = self.users.selectBy(prid=fr)
58         for i, user in enumerate(users):
59             user.prid = to
60         return fr, to,i+1 61
62     def delete(self):
63         rm = rrange(1,5)
64         users = self.users.selectBy(prid=rm)
65         for i, user in enumerate(users):
66             user.destroySelf()
67         return rm, i+1 68
69     def dbDump(self):
70         print '\n%s%s%s' % ('LOGIN'.ljust(COLSIZ),
71             'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
72         for usr in self.users.select():
73             print '%s%s%s' % (tuple([str(getattr(usr,
74                 field)).title().ljust(COLSIZ) \
75                 for field inFIELDS])) 76
77         drop = lambda self: self.users.dropTable()
78         finish = lambda self: self.cxn.close() 79
80     def main():
81         print '*** Connecting to %r database' % DBNAME
82         orm = MySQLObject('mysql',DBNAME) 83
84         print '\n*** Creating users table'
85         orm.create() 86
87         print '\n*** Inserting names into table'
88         orm.insert()
89         orm.dbDump() 90
91         print '\n*** Randomly moving folks',
92         fr, to, num = orm.update()
93         print 'from one group (%d) to another (%d)' % (fr,to)
94         print '\t(%d users moved)' % num
95         orm.dbDump() 96
97         print '\n*** Randomly choosing group',
98         rm, num = orm.delete()
99         print '(%d) to delete' % rm
100        print '\t(%d users removed)' % num
101        orm.dbDump()
102
103    print '\n*** Dropping userstable'
104    orm.drop()
105    orm.finish()
106

```

```
107 if name == '_main_':
108     main()
```

Line-by-Line Explanation Lines 110

As expected, we begin with module imports and constants. We follow the suggested style guideline of importing Python Standard Library modules first, followed by third-party or external modules, and finally, local modules to our application. The constants should be fairly self-explanatory.

Lines 1231

The constructor for our class, like `ushuffle_db.connect()`, does everything it can to make sure that there is a database available and returns a connection to it (lines 18-31). This is the only place you will see real SQL, as such activity is typically an operational task, not application-oriented.

Lines 3344

The `TRy-except` clause (lines 33-40) is used to reload an existing table or make a new one if it does not exist yet. Finally, we attach the relevant objects to our instance.

Lines 4670

These next four methods represent the core database functionality of table creation (lines 46 -52), insertion (lines 54-57), update (lines 59-64), and deletion (lines 66-70). We should also have a method for dropping the table:

```
def drop(self): self.users.drop()
```

or

```
drop = lambda self: self.users.drop()
```

However, we made a decision to give another demonstration of *delegation* (as introduced in [Chapter 13](#), Object-Oriented Programming). Delegation is where missing functionality (method call) is passed to another object in our instance which has it. See the explanation of lines 79-80.

Lines 7277

The responsibility of displaying proper output to the screen belongs to the `dbDump()` method. It extracts the rows from the database and pretty-prints the data just like its equivalent in `ushuffle_db.py`. In fact, they are nearly identical.

Lines 7980

We deliberately avoided creating a `drop()` method for the table since it would just call the table's drop method anyway. Also, there is no added functionality, so why create yet another function to have to maintain? The `getattr_()` special method is called whenever an attribute lookup fails.

If our object calls `orm.drop()` and finds no such method, `getattr(orm, 'drop')` is invoked. When that happens, `getattr_()` is called and delegates the attribute name to `self.users`. The interpreter will find that `self.users` has a `drop` attribute and pass that method call to it: `self.users.drop()`!

Lines 8284

The last method is `finish()`, which commits the transaction.

Lines 86114

The `main()` function drives our application. It creates a `MySQLAlchemy` object and uses that for all database operations. The script is the same as for our original application, `ushuffle_db.py`. You will notice that the database parameter `db` is optional and does not serve any purpose here in `ushuffle_sa.py` or the upcoming `SQLObject` version `ushuffle_so.py`. This is a placeholder for you to add support for other RDBMSs in these applications (see Exercises at the end of the chapter).

Upon running this script, you may get output that looks like this:

```
$ ushuffle_sa.py
```

```
*** Connecting to 'test' database
```

```
*** Creating users table
```

```
*** Inserting names into table
```

LOGIN	USERID	PROJ#
Serena	7003	4
Faye	6812	4
Leslie	7808	3
Ernie	7410	1
Dave	7306	2
Melissa	8602	1
Amy	7209	3
Angela	7603	4
Jess	7912	2
Larry	7311	1
Jim	7512	2
Davina	7902	3
Stan	7607	4
Pat	7711	2
Aaron	8312	2
Elliot	7911	3

```
*** Randomly moving folks from one group (1) to another(3)  
(3 users moved)
```

LOGIN	USERID	PROJ#
Serena	7003	4
Faye	6812	4
Leslie	7808	3
Ernie	7410	3
Dave	7306	2
Melissa	8602	3
Amy	7209	3
Angela	7603	4
Jess	7912	2
Larry	7311	3
Jim	7512	2
Davina	7902	3
Stan	7607	4
Pat	7711	2
Aaron	8312	2
Elliot	7911	3

```
*** Randomly choosing group (2) to delete (5 users removed)
```

LOGIN	USERID	PROJ#
Serena	7003	4
Faye	6812	4
Leslie	7808	3
Ernie	7410	3
Melissa	8602	3
Amy	7209	3
Angela	7603	4
Larry	7311	3
Davina	7902	3

```
Stan      7607      4
Elliot    7911      3
```

```
*** Dropping users table $
```

Line-by-Line Explanation Lines 110

This modules imports and constant declarations are practically identical to those of `ushuffle_sa.py` except that we are using `SQLObject` instead of `SQLAlchemy`.

Lines 1242

The constructor for our class does everything it can to make sure that there is a database available and returns a connection to it, just like our `SQLAlchemy` example. Similarly, this is the only place you will see real SQL. Our application, as coded here, will result in an infinite loop if for some reason a `Users` table cannot be created in `SQLObject`.

We are trying to be clever in handling errors by fixing the problem and retrying the table (re)create. Since `SQLObject` uses metaclasses, we know that special magic is happening under the covers, so we have to define two different classes one for if the table already exists and another if it does not.

The code works something like this:

1.

Try and establish a connection to an existing table; if it works, we are done (lines 23-29)

2.

Otherwise, create the class from scratch for the table; if so, we are done (lines 31-36)

3.

Otherwise, we have a database issue, so try and make a new database (lines 37-40)

4.

Loop back up and try all this again

Hopefully it (eventually) succeeds in one of the first two places. When the loop is terminated, we attach the relevant objects to our instance as we did in `ushuffle_sa.py`.

Lines 4467, 7778

The database operations happen in these lines. We have table create (lines 44-47) and drop (line 77), insert (lines 49-52), update (lines 54-60), and delete (lines 62-67). The `finish()` method on line 78 is to close the connection. We could not use delegation for table drop like we did for the `SQLAlchemy` example because the would-be delegated method for it is called `dropTable()` not `drop()`.

Lines 6975

This is the same and expected `dbDump()` method, which pulls the rows from the database and displays things nicely to the screen.

Lines 80108

This is the `main()` function again. It works just like the one in `ushuffle_sa.py`. Also, the `db` argument to the constructor is a placeholder for you to add support for other RDBMSs in these applications (see Exercises at the end of the chapter).

Here is what your output may look like if you run this script:

```
$ ushuffle_so.py
```

```
*** Connecting to 'test' database
```

```
*** Creating users table
```

```
*** Inserting names into table LOGIN USERID      PROJ#
```

```
      Jess      7912      1
```

Amy	7209	4
Melissa	8602	2
Dave	7306	4
Angela	7603	4
Serena	7003	2
Aaron	8312	1
Leslie	7808	1
Stan	7607	3
Pat	7711	3
Jim	7512	4
Larry	7311	3
Ernie	7410	2
Faye	6812	4
Davina	7902	1
Elliot	7911	4

*** Randomly moving folks from one group (2) to another (3) (3 users moved)

LOGIN	USERID	PROJ#
Jess	7912	1
Amy	7209	4
Melissa	8602	3
Dave	7306	4
Angela	7603	4
Serena	7003	3
Aaron	8312	1
Leslie	7808	1
Stan	7607	3
Pat	7711	3
Jim	7512	4
Larry	7311	3
Ernie	7410	3
Faye	6812	4
Davina	7902	1
Elliot	7911	4

*** Randomly choosing group (3) to delete
(6 users removed)

LOGIN	USERID	PROJ#
Jess	7912	1
Amy	7209	4
Dave	7306	4
Angela	7603	4
Aaron	8312	1
Leslie	7808	1
Jim	7512	4
Faye	6812	4
Davina	7902	1
Elliot	7911	4

*** Dropping users table

\$

Related Modules

[Table 21.8](#) lists most of the common databases out there along with working Python modules and packages that serve as adapters to those database systems. Note that not all adapters are DB-API- compliant.

Table 21.8. Database-Related Modules and Websites

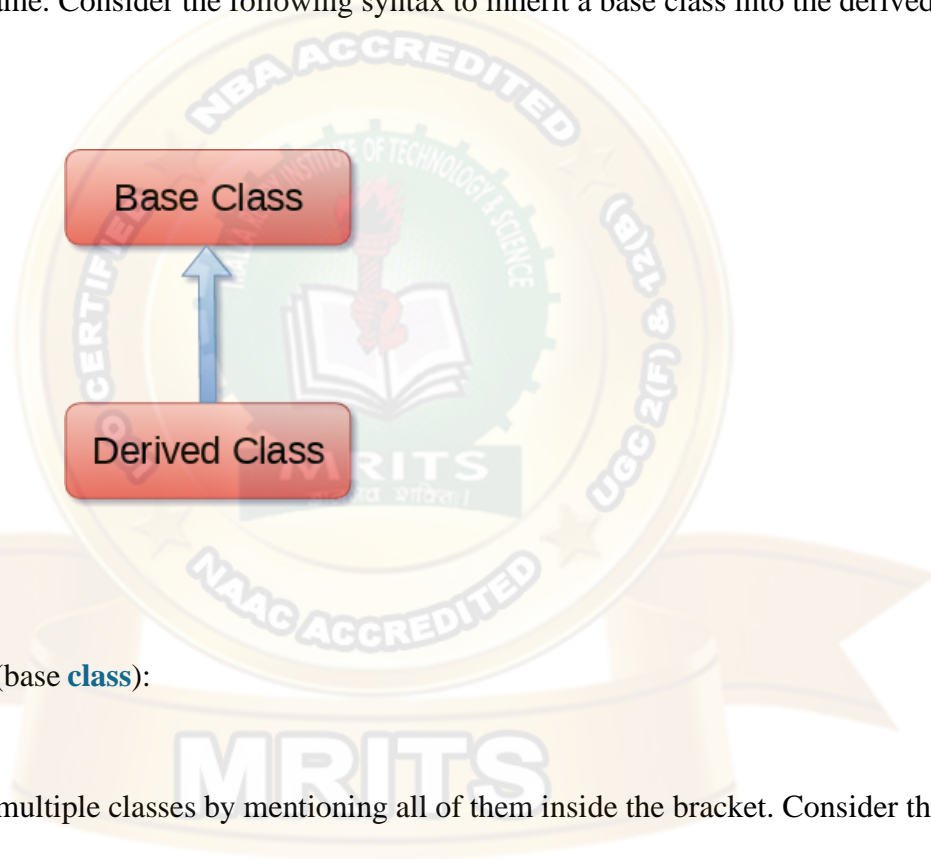
Name	Online Reference or Description
Databases	
Gadfly	http://gadfly.sf.net
MySQL	http://mysql.com or http://mysql.org
MySQLdb a.k.a. MySQL-python	http://sf.net/projects/mysql-python
PostgreSQL	http://postgres.org
psycopg	http://initd.org/projects/psycopa1
psycopg2	http://initd.org/software/initd/psycopa/
PyPgSQL	http://pypqsql.sf.net
PyGreSQL	http://pygresql.org
PoPy	Deprecated; merged into PyGreSQL project
SQLite	http://sqlite.org
pysqlite	http://initd.org/projects/pysqlite
<code>sqlite3</code> ^[a]	<code>pysqlite</code> integrated into Python Standard Library; use this one unless you want to download the latest patch
APSW	http://rogerbinns.com/apsw.html
MaxDB (SAP)	http://mysql.com/products/maxdb
sdb	http://dev.mysql.com/downloads/maxdb/7.6.00.html#Python
sapdb	http://sapdb.org/sapdbPython.html
Firebird (InterBase)	http://firebird.sf.net
KInterbasDB	http://kinterbasdb.sf.net
SQL Server	http://microsoft.com/sql
pymssql	http://pymssql.sf.net (requires FreeTDS [http://freetds.org])
adodbapi	http://adodbapi.sf.net
Sybase	http://sybase.com
sybase	http://object-craft.com.au/projects/sybase
Oracle	http://oracle.com
cx_Oracle	http://starship.python.net/crew/atuning/cx_Oracle
DCOracle2	http://zope.org/Members/matt/dco2 (older, for Oracle8 only)
Ingres	http://ingres.com
Ingres DBI	http://ingres.com/products/ Prod_Download_Python_DBI.html
ingmod	http://www.informatik.uni-rostock.de/~hme/software/
ORMs	
SQLObject	http://sqlobject.org
SQLAlchemy	http://sqlalchemy.org
PyDO/PyDO2	http://skunkweb.sf.net/pydo.html

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



Syntax

```
class derived-class(base class):  
    <class-suite>
```

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Example

```
class Animal:  
    def speak(self):  
        print("Animal Speaking")  
  
#child class Dog inherits the base class Animal  
class Dog(Animal):
```

```
def bark(self):  
    print("dog barking")  
  
d = Dog()  
  
d.bark()  
  
d.speak()
```

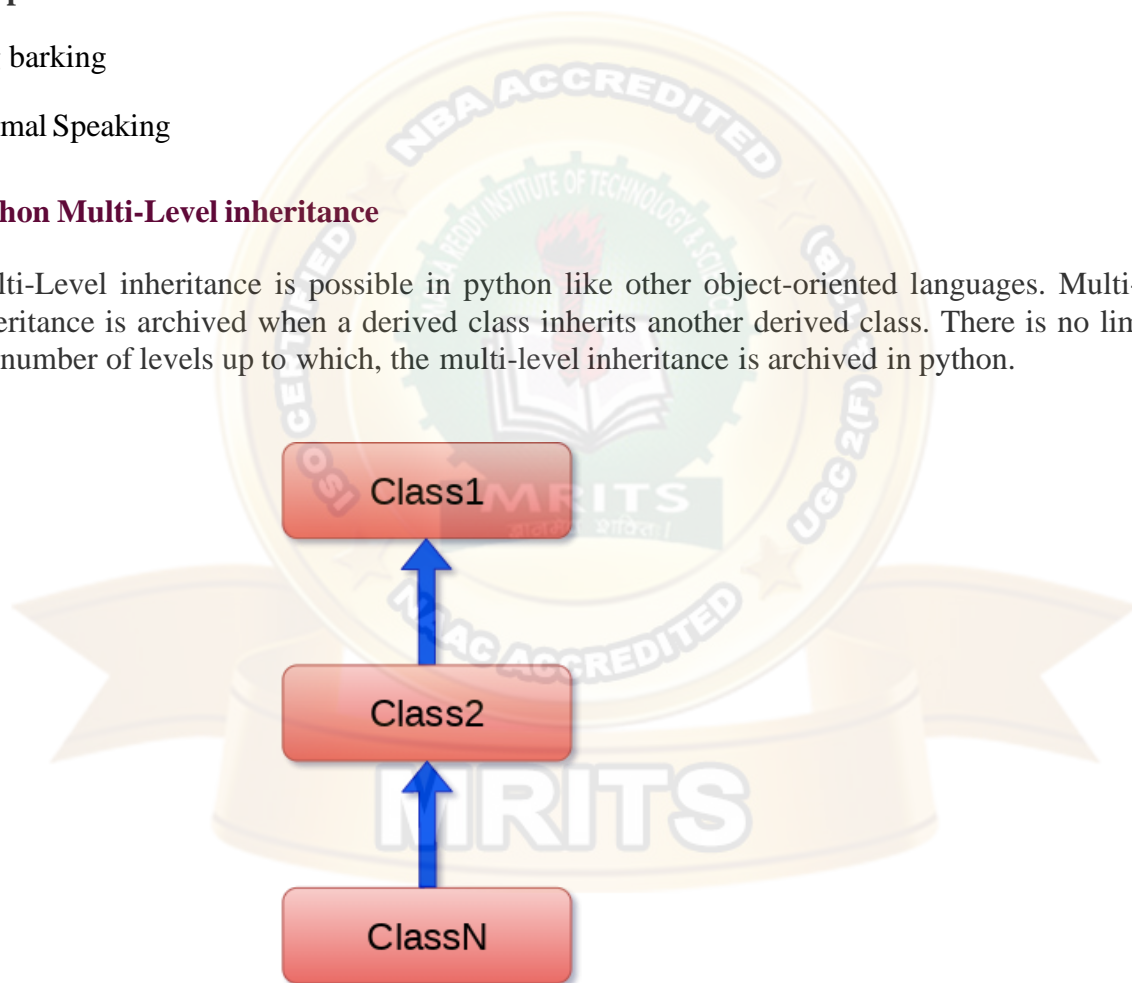
Output

dog barking

Animal Speaking

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



Syntax

```
class class1:  
    <class-suite>  
  
class class2(class1):
```

```
<class suite>
```

```
class class3(class2):
```

```
<class suite>
```

```
.
```

Example

```
class Animal:
```

```
    def speak(self):
```

```
        print("Animal Speaking")
```

```
#The child class Dog inherits the base class Animal
```

```
class Dog(Animal):
```

```
    def bark(self):
```

```
        print("dog barking")
```

```
#The child class Dogchild inherits another child class Dog
```

```
class DogChild(Dog):
```

```
    def eat(self):
```

```
        print("Eating bread...")
```

```
d = DogChild()
```

```
d.bark()
```

```
d.speak()
```

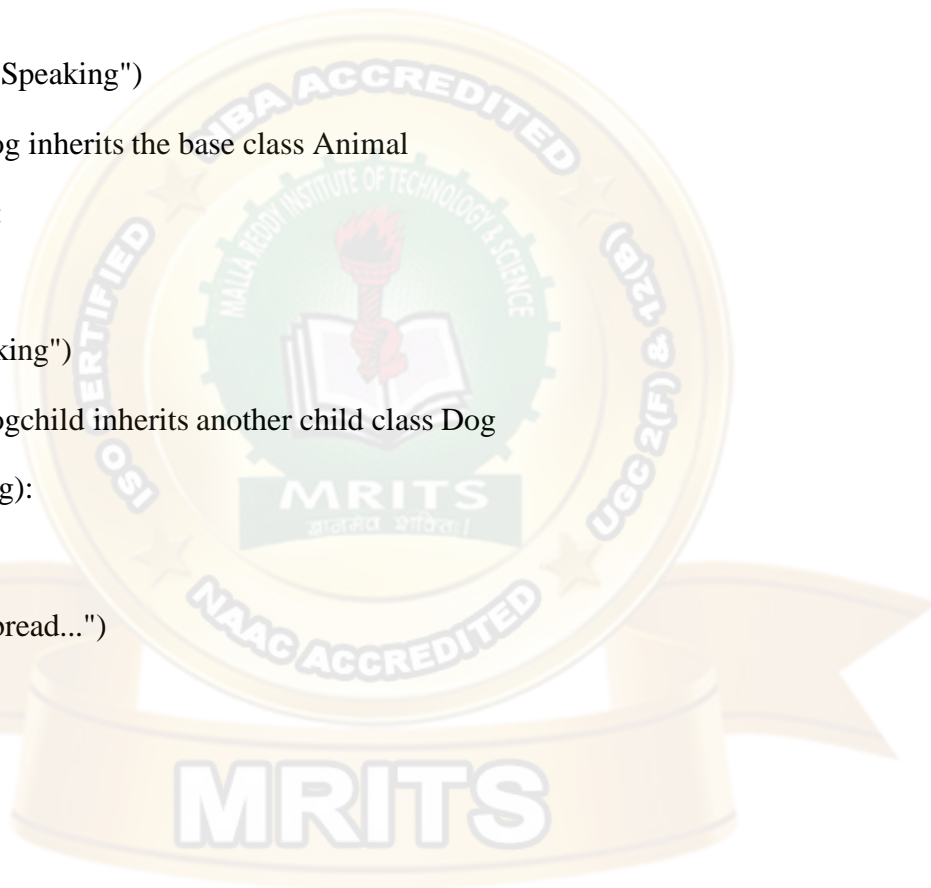
```
d.eat()
```

Output:

```
dog barking
```

```
Animal Speaking
```

```
Eating bread...
```



Python JSON

JSON stands for JavaScript Object Notation, which is a widely used data format for data interchange on the web. JSON is the ideal format for organizing data between a client and a server. Its syntax is similar to the JavaScript programming language. The main objective of JSON is to transmit the data between the client and the web server. It is easy to learn and the most effective way to interchange the data. It can be used with various programming languages such as Python, Perl, Java, etc.

JSON mainly supports 6 types of data type In JavaScript:

- String
- Number
- Boolean
- Null
- Object
- Array

JSON is built on the two structures:

It stores data in the name/value pairs. It is treated as an object, record, dictionary, hash table, keyed list.

The ordered list of values is treated as an array, vector, list, or sequence.

JSON data representation is similar to the Python dictionary. Below is an example of JSON data:

```
{
  "book": [
    {
      "id": 01,
      "language": "English",
      "edition": "Second",
      "author": "Derrick Mwiti"  ],

```



```
{  
  
{  
    "id": 02,  
    "language": "French",  
    "edition": "Third",  
    "author": "Vladimir"  
}  
}
```

Working with Python JSON

Python provides a module called json. Python supports standard library marshal and pickle module, and JSON API behaves similarly as these library. Python natively supports JSON features.

The encoding of JSON data is called Serialization. Serialization is a technique where data transforms in the series of bytes and transmitted across the network.

The deserialization is the reverse process of decoding the data that is converted into the JSON format.

This module includes many built-in functions.

```
import json  
print(dir(json))
```

Output:

```
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtin__']
```

Writing JSON Data into File

Python provides a dump() function to transmit(encode) data in JSON format. It accepts two positional arguments, first is the data object to be serialized and second is the file-like object to which the bytes needs to be written.

serialization example:

```
Import json

# Key:value mapping

student = {

    "Name" : "Peter",

    "Roll_no" : "0090014",

    "Grade" : "A",

    "Age": 20,

    "Subject": ["Computer Graphics", "Discrete Mathematics", "Data Structure"]

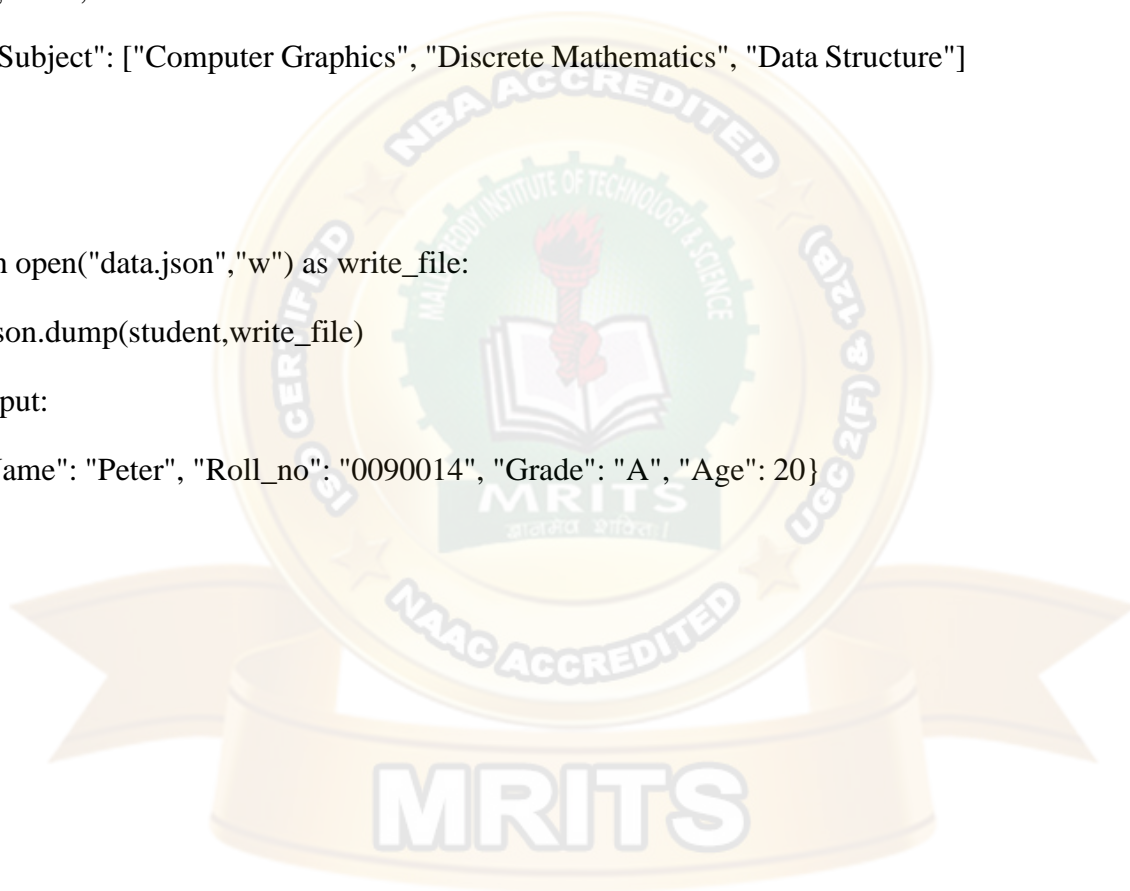
}

with open("data.json","w") as write_file:

    json.dump(student,write_file)
```

Output:

```
{"Name": "Peter", "Roll_no": "0090014", "Grade": "A", "Age": 20}
```





MALLA REDDY INSTITUTE OF TECHNOLOGY AND SCIENCE

(Sponsored by Malla Reddy Educational Society)

Permanently affiliated to JNTUH Hyderabad and Approved by AICTE, New Delhi
NAAC & NBA Accredited, ISO 9001:2015 Certified, Approved by U K Accreditation centre.

Granted status of 2 (f) and 12(b) under UGC Act 1956, Govt. of India.

Maisammaguda, Dhulapally, Secunderabad – 500100.



UNIT-I

1. Which of the following is not in Python Character Set.

- a. Letters : A-Z or a – z
- b. Digits : 0 – 9
- c. Whitespaces : blank space, tab etc

d. Images : Vector

2. Which of the following is not the mode of interacting with python?

- a. Interactive Mode
- b. Script Mode
- c. Hybrid Mode**
- d. None of the above

3. Python supports dynamic typing.

- a. True**
- b. False

4. What will be the data type of the following variable?

A = '101'

- a. Integer
- b. String**
- c. Float
- d. None of the above

5. Write the output of the following:

```
print(range(0,8,2))
```

- a. 0,2,4,6**
- b. range(0, 8, 2)
- c. Error
- d. None of the above

6. Which of the following is not correct about python?

- a. Python is an open source language.
- b. Python is based on ABC language.
- c. Python is developed by Guido Van Rossum
- d. None of the above**

7. Smallest element of of python coding is called _____

- a. Identifiers
- b. Token**
- c. Keywords
- d. Delimiters

8. Which of the following is not a token?

- a. //
- b. "X"
- c. ##**
- d. 23

9. Write the output of the following code:

```
x=2
x=5
x=x+x
print(x)
```

- a. 7
- b. 4
- c. 10**
- d. Error

10. Write the output of the following code :

```
x=2
y=3
x+y+5
print(x+y)
```

- a. 10



- b. 5
- c. Error
- d. None of the above

11. Which of the following symbol is used to write comment?

- a. ?
- b. //
- c. #
- d. **

12. Writing comments is mandatory in python programs(T/F)

a. True

b. False

13. Each statement in python is terminated by _____

- a. Semicolon(;
- b. Colon(:)
- c. Comma(,)

d. None of the above

14. Write the output of the following:

```
print('Hello, world!');print("H")
```

- a. Hello world H
- b. Hello worldH

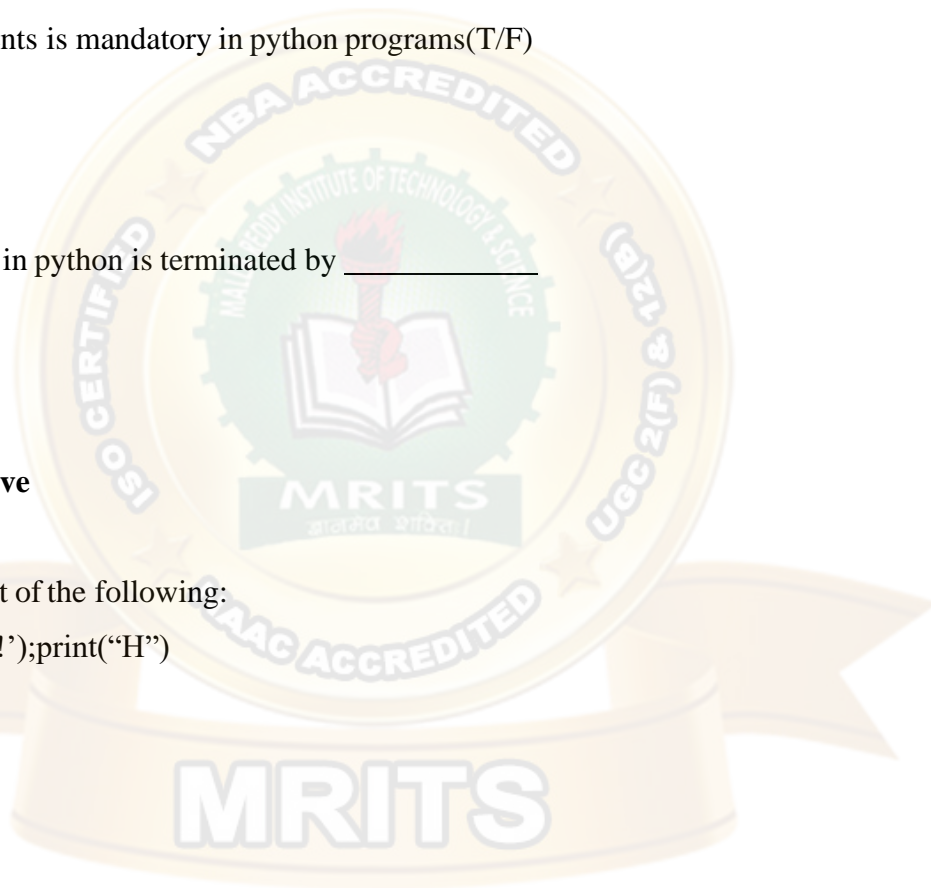
c. Hello world

H

d. Error

15. _____spaces should be left for indentation.

- a. 3
- b. 4**
- c. 5
- d. 0



16. What type of error is returned by the following statement?

```
def abc(a):
```

```
    print(a)
```

a. `ErrorIndentation`

b. `IndentationError`

c. `SpaceError`

d. No error in the given statement

17. return statement is mandatory in function definition.(T/F)

a. True

b. False

18. Which keyword is used to define a function in python?

a. `def`

b. `define`

c. `new`

d. None of the above

19. Write the output of the following:

```
a=8
```

```
def abc(a):
```

```
    print(a)
```

```
abc(7)
```

a. 8

b. Error

c. No Output

d. 7

20. Statement below “function definition” begin with spaces called_____

a. Indentation

b. Condition

c. Definition

d. None of the above



21. Which of the following is invalid variable name?

- a. Sum1
- b. Num_1
- c. Num 1**
- d. N1

22. Write the output of the following.

```
def abc():
```

```
    print("abc")
```

- a. abc
- b. Error
- c. 0
- d. No Output**

23. Python is case sensitive.(T/F)

- a. True**
- b. False

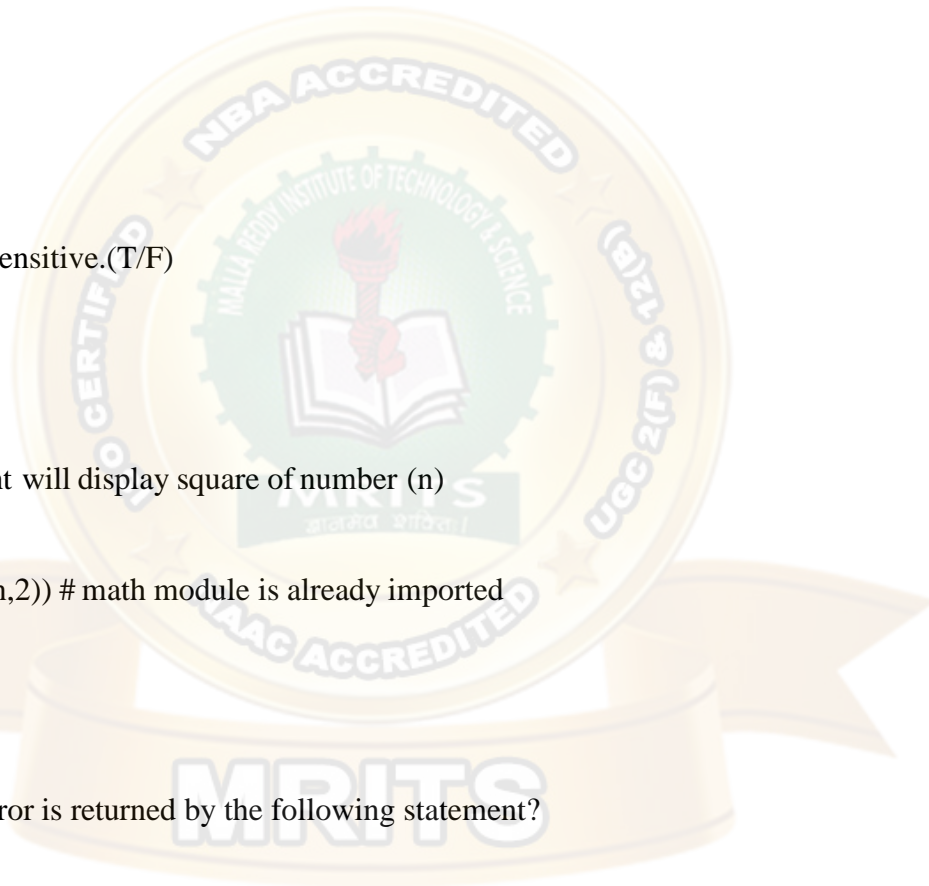
24. Which statement will display square of number (n)

- a. `print(n * n)`
- b. `print(math.pow(n,2))` # math module is already imported
- c. All of the above**
- d. Only First

25. What type of error is returned by the following statement?

```
print(eval(13))
```

- a. `SyntaxError`
- b. `TypeError`**
- c. `ValueError`
- d. No Error in this statement



26. Which statement is adding remainder of 8 divided by 3 to the product of 5 and 6?

a. $8 \% 3 + 5 * 6$

b. $8/3 + 5 * 6$

c. $8 // 3 + 6.5$

d. None of the above

27. Is `a,b = 6` statement will return an error.(T/F)

a. True

b. False

28. Identify the invalid identifier.

a. Keyword

b. token

c. operator

d. and

29. Both the print statement will return same output (T/F)

`a=9`

`b=a`

`print(id(a))`

`print(id(b))`

a. False

b. True

30. Which keyboard key is used to run python programs?

a. F6

b. F5

c. F + n

d. Ctrl + r

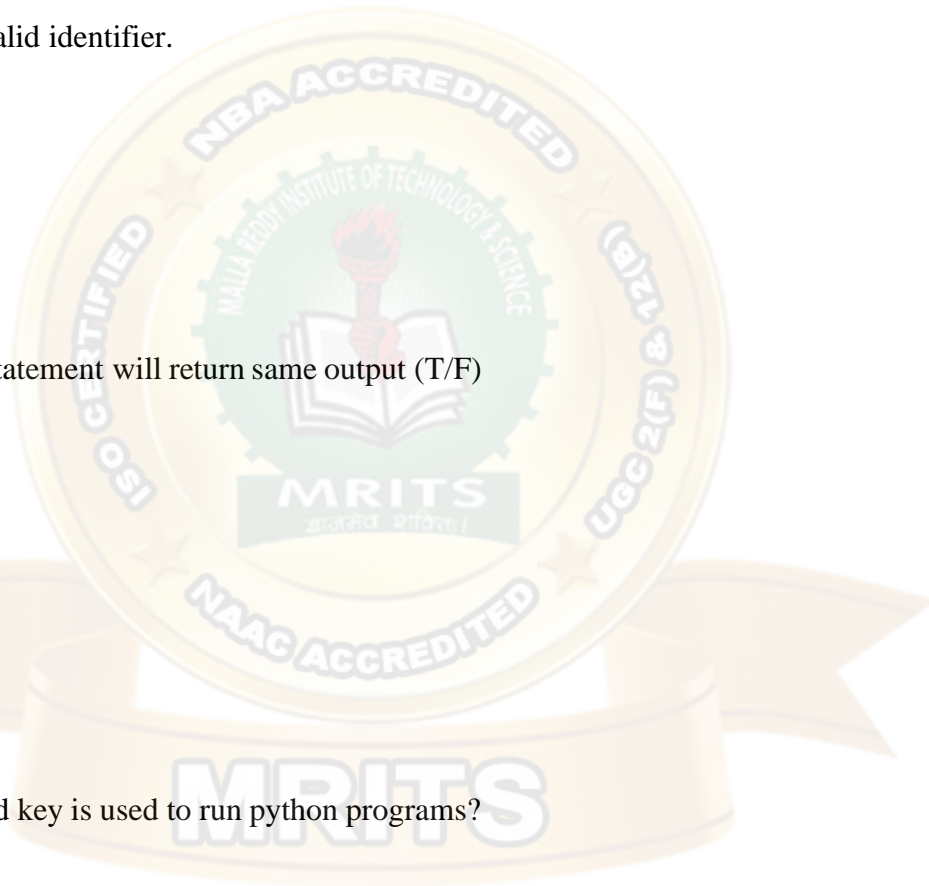
31. Which method is used to find the memory location of variable?

a. `id()`

b. `add()`

c. `type()`

d. None of the above



32. _____method is used to find the data type of a variable.

- a. `type()`
- b. `dtype()`
- c. `typed()`
- d. None of the above

33. _____escape sequence is used for horizontal tab.

- a. `\n`
- b. `\t`
- c. `\T`
- d. No

34. An escape sequence is represented by _____slash followed by one or two characters.

- a. back
- b. forward
- c. double
- d. None of the above

35. Write the output of the following code :

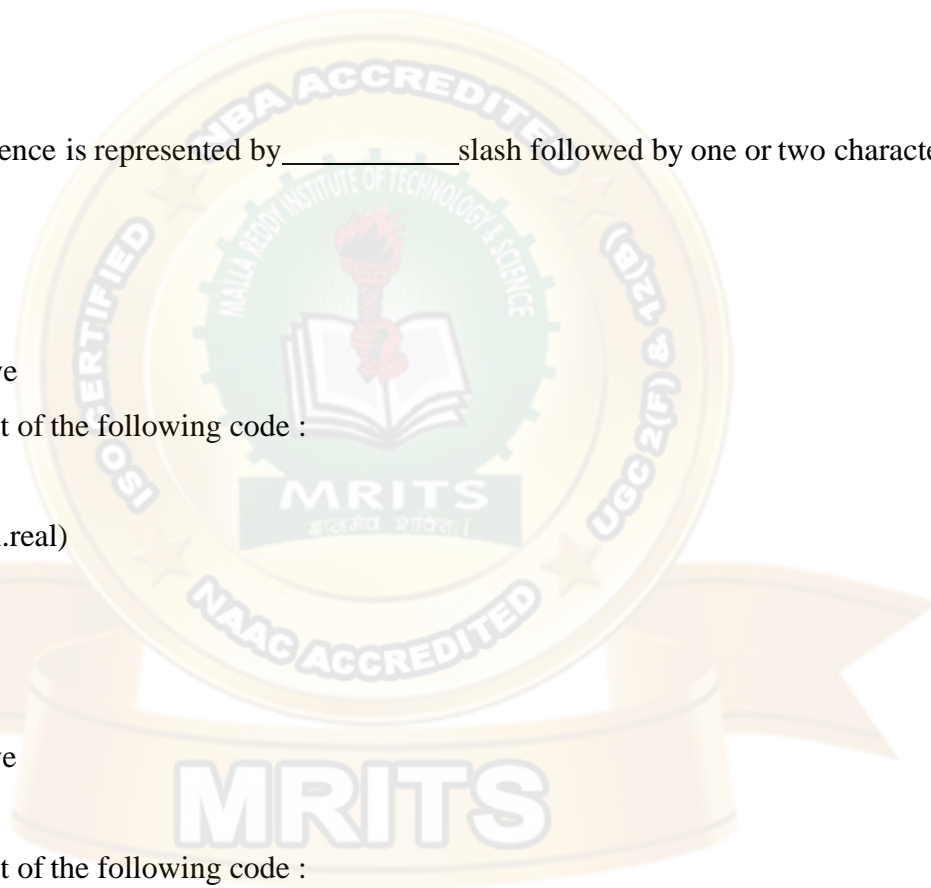
```
>>> x = 4 - 7j
>>> print(x.imag, x.real)
```

- a. 4.0 -7.0
- b. -7.0 4.0
- c. Error
- d. None of the above

36. Write the output of the following code :

```
>>> a=9
>>> x=str(a)
>>> b=5
>>> y=str(b)
>>> x+y
```

- a. 14
- b. 9,5
- c. 95
- d. None of the above



37. Write the output of the following code :

```
>>> 7+2//1**2 > 5+2**2//3
```

- a. True
- b. False
- c. Error
- d. None of the above

38. Write the output of the following code :

```
>>> s = None
```

```
>>> s
```

- a. Nothing will be printed
- b. None
- c. Shows Error
- d. None of the above

39. Which of the following assignment will return error?

- a. a = b = c = 89
- b. a = 6, b = 8
- c. a, b, c = 1, 2, 3
- d. None of the above

40. Which of the following is wrong in reference to naming of variable?

- a. Keywords are not allowed for variable names.
- b. Spaces are not allowed for variable names.
- c. Variable names can start from number.
- d. Special symbols are not allowed

41. Which of the following is invalid identifier?

- a. _
- b. _1st
- c. 1stName
- d. While

42. Identifier name can be of maximum_____ character

- a. 63
- b. 79
- c. 53
- d. any number of

43. Which of the following can not be used as an identifier?

- a. eval
- b. max
- c. **pass**
- d. All of the above

44. All keywords in Python are in lower case(T/F).

- a. True
- b. **False**

45. Which of the following statement is calculating x raise to power n?

- a. $x * n$
- b. **$x ** n$**
- c. $n ** x$
- d. $x ^ n$

46. Write the output of the following.

```
m, n, p = 1, 2, 3
```

```
print(m, n, p)
```

- a. 1, 2, 3
- b. **1 2 3**
- c. Print 1, 2 and 3 vertically
- d. Error

47. Which of the following is valid operator?

- a. **in**
- b. on
- c. it
- d. at



48. Output of `print(7 % 21)` is _____

- a. 3
- b. 7
- c. None of the above
- d. Error

49. Operators of same precedence are executed from _____

- a. **left to right**
- b. right to left
- c. in any order
- d. None of the above

50. Output of `print(2 * 3 ** 2)` is

- a. 16
- b. 64
- c. **18**
- d. Error

51. Write the output of the following:

```
x = int(7) + int('9')
```

```
print(x)
```

- a. 79
- b. **16**
- c. Error
- d. None of the above

52. Out of addition(+) and Subtraction (-) operator, which has more precedence?

- a. Addition (+)
- b. Subtraction (-)
- c. **Both have same precedence**
- d. None of the above

53. Which of the following statement will return error when $x = 7$?

- a. `print(x)`
- b. `print(int(x))`



c. print(eval(x))

d. None of the above

54. Which of the following store data in pair?

a. List

b. Tuple

c. String

d. Dictionary

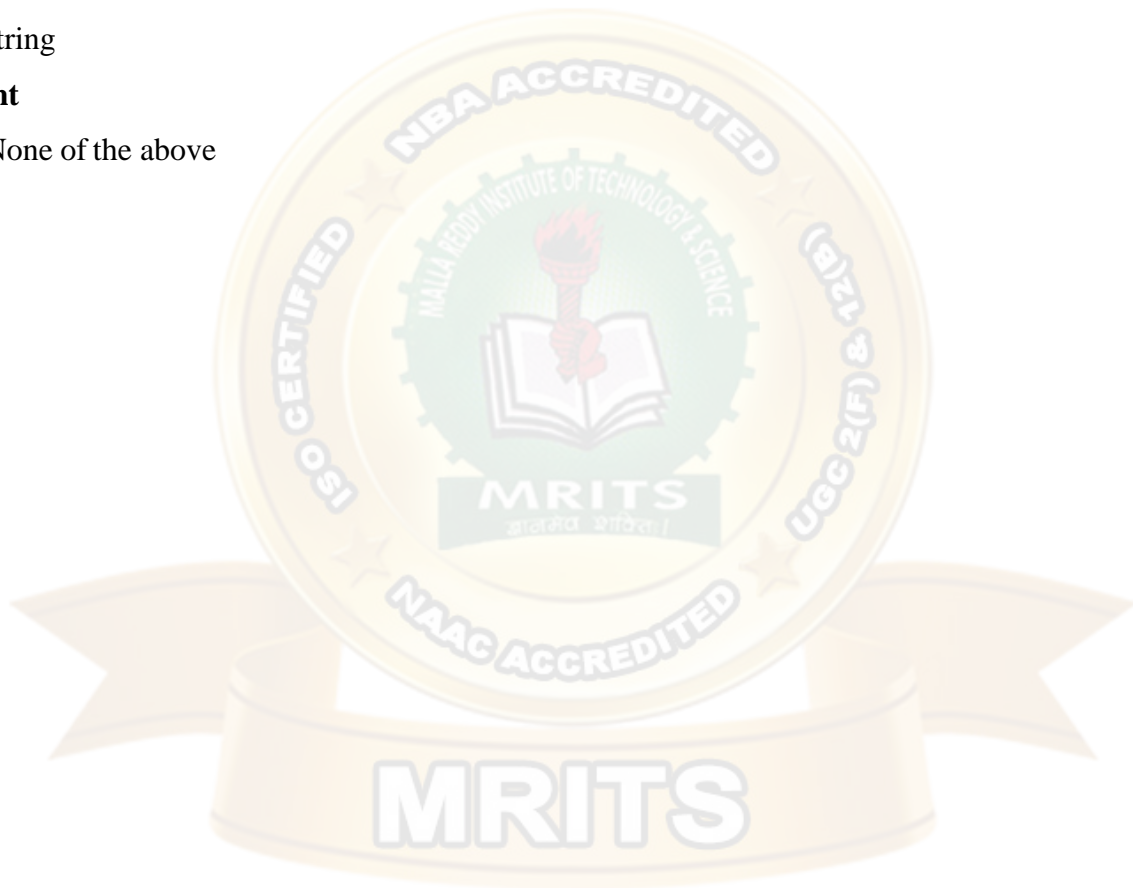
55. What is the return type of function id()?

a. float

b. string

c. int

d. None of the above



UNIT-II

1. To open a file c:\scores.txt for reading, we use _____

- a) `infile = open("c:\scores.txt", "r")`
- b) `infile = open("c:\\scores.txt", "r")`**
- c) `infile = open(file = "c:\scores.txt", "r")`
- d) `infile = open(file = "c:\\scores.txt", "r")`

2. To open a file c:\scores.txt for writing, we use _____

- a) `outfile = open("c:\scores.txt", "w")`
- b) `outfile = open("c:\\scores.txt", "w")`**
- c) `outfile = open(file = "c:\scores.txt", "w")`
- d) `outfile = open(file = "c:\\scores.txt", "w")`

3. To open a file c:\scores.txt for appending data, we use _____

- a) `outfile = open("c:\\scores.txt", "a")`**
- b) `outfile = open("c:\\scores.txt", "rw")`
- c) `outfile = open(file = "c:\scores.txt", "w")`
- d) `outfile = open(file = "c:\\scores.txt", "w")`

4. Which of the following statements are true?

- a) When you open a file for reading, if the file does not exist, an error occurs
- b) When you open a file for writing, if the file does not exist, a new file is created
- c) When you open a file for writing, if the file exists, the existing file is overwritten with the new file
- d) All of the mentioned**

5. To read two characters from a file object infile, we use _____

- a) `infile.read(2)`**
- b) `infile.read()`
- c) `infile.readline()`
- d) `infile.readlines()`

6. To read the entire remaining contents of the file as a string from a file object infile, we use _____

- a) `infile.read(2)`
- b) `infile.read()`**
- c) `infile.readline()`
- d) `infile.readlines()`

7. What will be the output of the following Python code?

```
f = None
for i in range (5):
    with open("data.txt", "w") as f:
        if i > 2:
            break
print(f.closed)
```

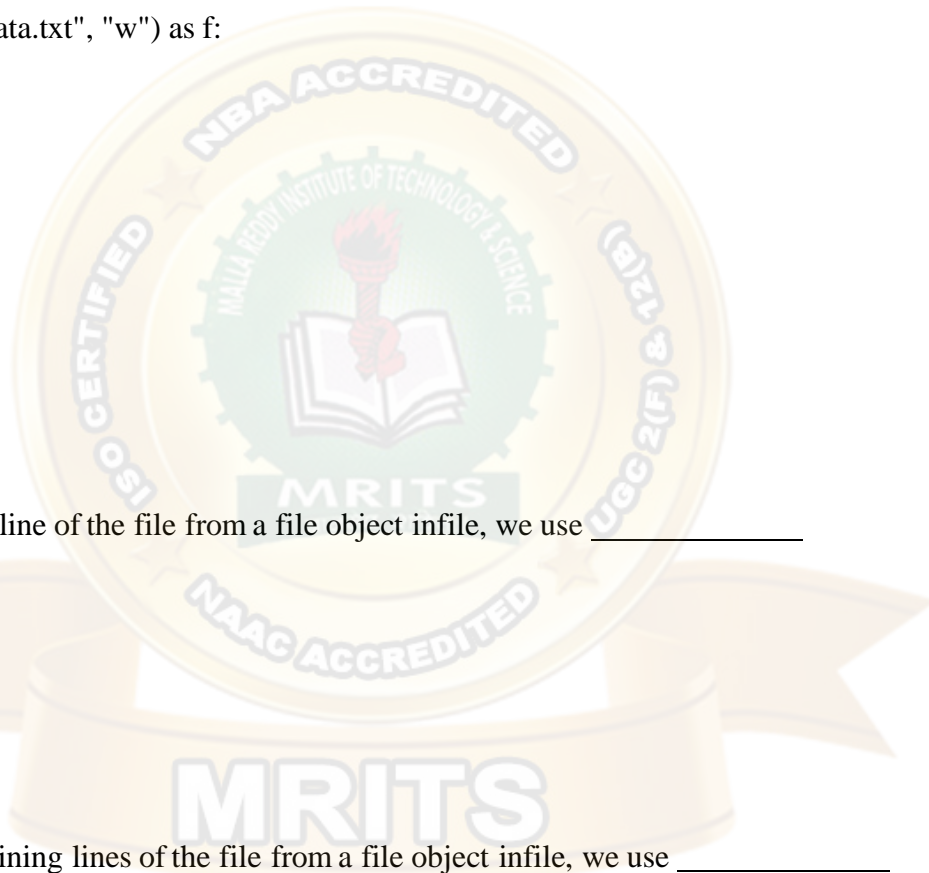
- a) True**
- b) False
- c) None
- d) Error

8. To read the next line of the file from a file object infile, we use _____

- a) `infile.read(2)`
- b) `infile.read()`
- c) `infile.readline()`**
- d) `infile.readlines()`

9. To read the remaining lines of the file from a file object infile, we use _____

- a) `infile.read(2)`
- b) `infile.read()`
- c) `infile.readline()`
- d) `infile.readlines()`**



10. The readlines() method returns _____

a) str

b) a list of lines

c) a list of single characters

d) a list of integers

11. How many except statements can a try-except block have?

a) zero

b) one

c) more than one

d) more than zero

12. When will the else part of try-except-else be executed?

a) always

b) when an exception occurs

c) when no exception occurs

d) when an exception occurs in to except block

13. Is the following Python code valid?

try:

 # Do something

except:

 # Do something

finally:

 # Do something

a) no, there is no such thing as finally

b) no, finally cannot be used with except

c) no, finally must come before except

d) yes

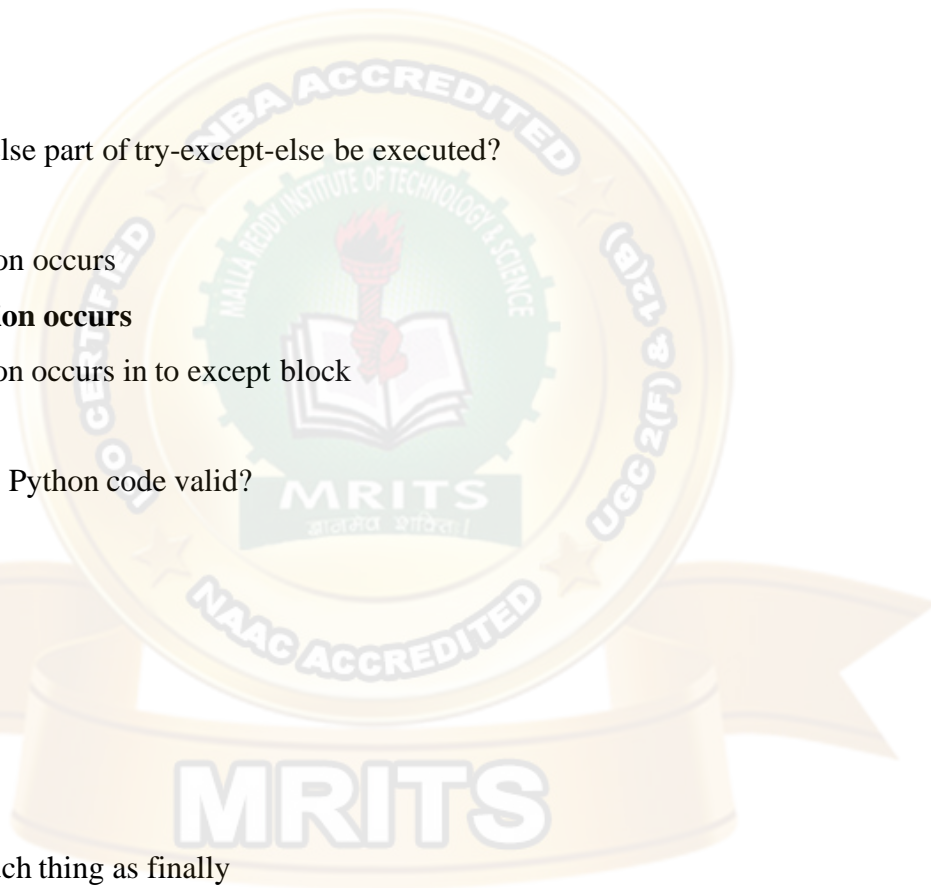
14. Is the following Python code valid?

try:

Do something

except:

Do something



else:

Do something

- a) no, there is no such thing as else
- b) no, else cannot be used with except
- c) no, else must come before except
- d) yes**

15. Can one block of except statements handle multiple exception?

- a) yes, like except TypeError, SyntaxError [...]**
- b) yes, like except [TypeError, SyntaxError]
- c) no
- d) none of the mentioned**

16. When is the finally block executed?

- a) when there is no exception
- b) when there is an exception
- c) only if some condition that has been specified is satisfied
- d) always**

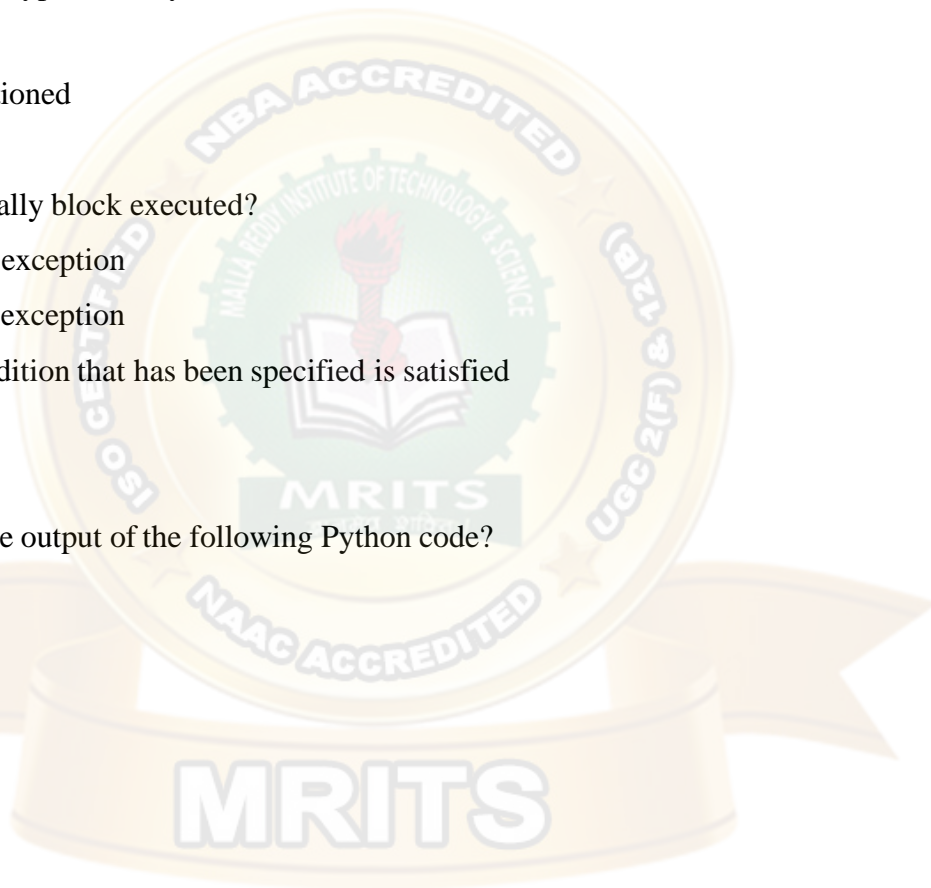
17. What will be the output of the following Python code?

```
def foo():  
    try:  
        return 1  
    finally:  
        return 2
```

```
k = foo()
```

```
print(k)
```

- a) 1
- b) 2**
- c) 3
- d) error, there is more than one return statement in a single try-finally block



18. What will be the output of the following Python code?

```
def foo():  
    try:  
        print(1)  
    finally:  
        print(2)
```

foo()

- a) 1 2
- b) 1
- c) 2
- d) none of the mentioned

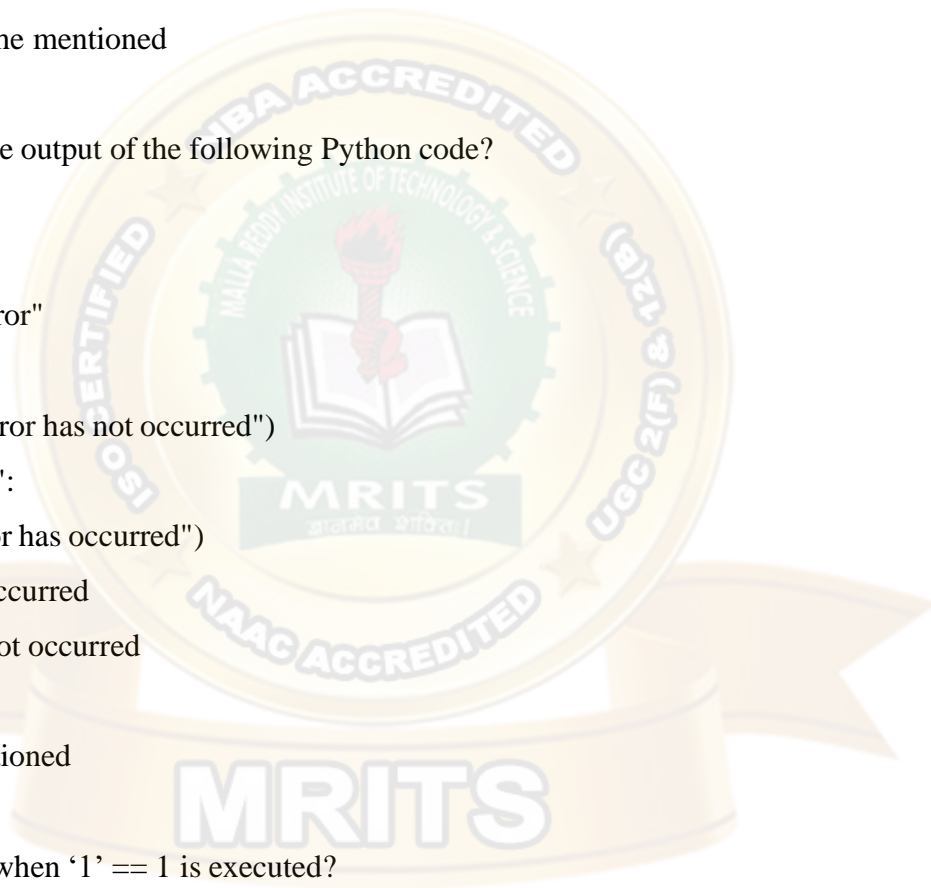
19. What will be the output of the following Python code?

```
try:  
    if '1' != 1:  
        raise "someError"  
    else:  
        print("someError has not occurred")  
except "someError":  
    print ("someError has occurred")
```

- a) someError has occurred
- b) someError has not occurred
- c) **invalid code**
- d) none of the mentioned

20. What happens when '1' == 1 is executed?

- a) we get a True
- b) **we get a False**
- c) an TypeError occurs
- d) a ValueError occurs



21. The following Python code will result in an error if the input value is entered as -5.

```
assert False, 'Spanish'
```

a) **True**

b) False

22. What will be the output of the following Python code?

```
x=10
```

```
y=8
```

```
assert x>y, 'X too small'
```

a) Assertion Error

b) 10 8

c) **No output**

d) 108

23. What will be the output of the following Python code?

```
#generator
```

```
def f(x):
```

```
    yield x+1
```

```
g=f(8)
```

```
print(next(g))
```

a) 8

b) **9**

c) 7

d) Error

24. What will be the output of the following Python code?

```
def f(x):
```

```
    yield x+1
```

```
    print("test")
```

```
    yield x+2
```

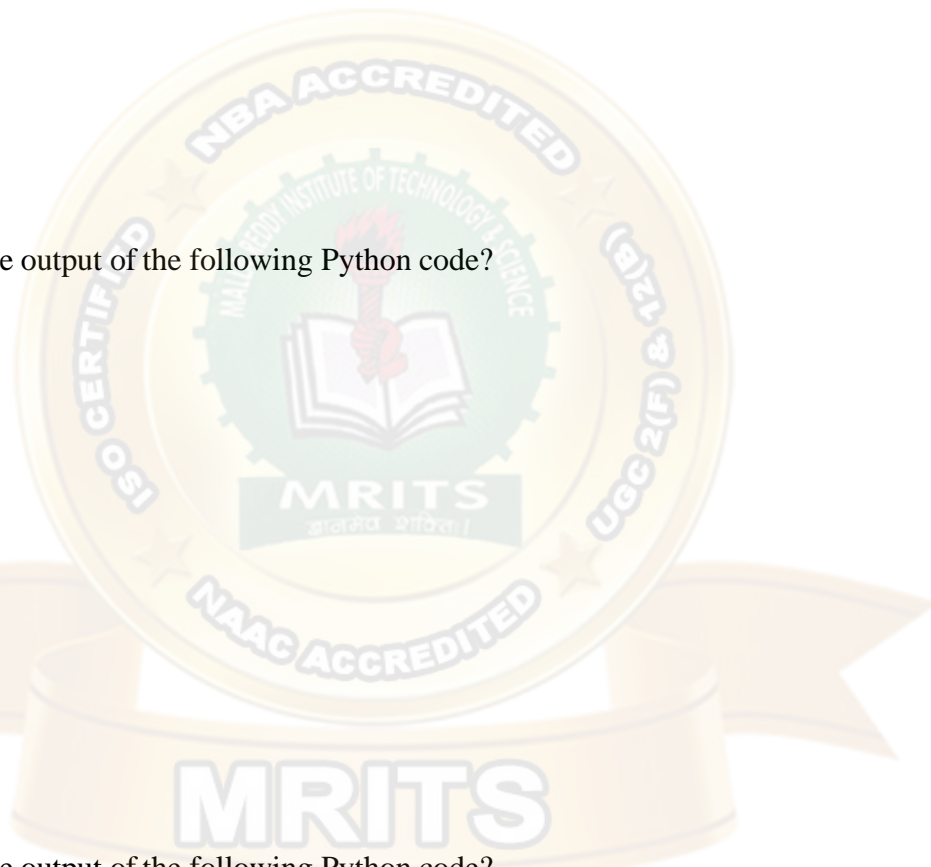
```
g=f(9)
```

a) Error

b) test

c) test

10



12

d) No output

25. What will be the output of the following Python code?

```
def f(x):
```

```
    yield x+1
```

```
    print("test")
```

```
    yield x+2
```

```
g=f(10)
```

```
print(next(g))
```

```
print(next(g))
```

a) No output

b) 11

test

12

c) 11

test

d) 11

26. What will be the output of the following Python code?

```
def a():
```

```
    try:
```

```
        f(x, 4)
```

```
    finally:
```

```
        print('after f')
```

```
        print('after f?')
```

a()

a) No output

b) after f?

c) error

d) after f



27. What will be the output of the following Python code?

```
def f(x):
```

```
    for i in range(5):
```

```
        yield i
```

```
g=f(8)
```

```
print(list(g))
```

a) [0, 1, 2, 3, 4]

b) [1, 2, 3, 4, 5, 6, 7, 8]

c) [1, 2, 3, 4, 5]

d) [0, 1, 2, 3, 4, 5, 6, 7]

28. Which of these definitions correctly describes a module?

a) Denoted by triple quotes for providing the specification of certain program elements

b) Design and implementation of specific functionality to be incorporated into a program

c) Defines the specification of how it is to be used

d) Any program that reuses code

29. Which of the following is not an advantage of using modules?

a) Provides a means of reuse of program code

b) Provides a means of dividing up tasks

c) Provides a means of reducing the size of the program

d) Provides a means of testing individual parts of the program

30. Program code making use of a given module is called a _____ of the module.

a) **Client**

b) Docstring

c) Interface

d) Modularity

31. _____ is a string literal denoted by triple quotes for providing the specifications of certain program elements.

a) Interface

b) Modularity

c) Client

d) Docstring

32. Which of the following isn't true about main modules?

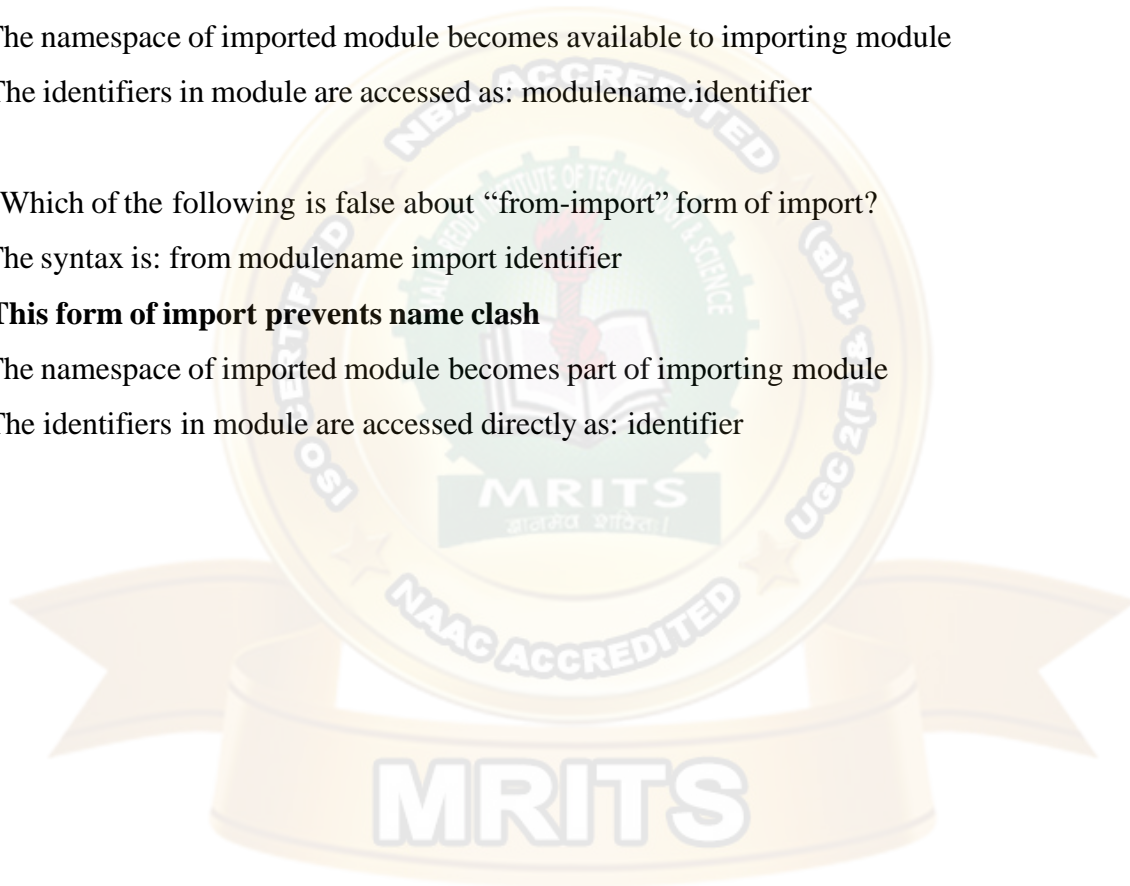
- a) When a python file is directly executed, it is considered main module of a program
- b) Main modules may import any number of modules
- c) Special name given to main modules is: `__main__`
- d) Other main modules can import main modules**

33. Which of the following is false about "import modulename" form of import?

- a) The namespace of imported module becomes part of importing module**
- b) This form of import prevents name clash
- c) The namespace of imported module becomes available to importing module
- d) The identifiers in module are accessed as: `modulename.identifier`

34. Which of the following is false about "from-import" form of import?

- a) The syntax is: `from modulename import identifier`
- b) This form of import prevents name clash**
- c) The namespace of imported module becomes part of importing module
- d) The identifiers in module are accessed directly as: `identifier`



UNIT-III

1. Python has a built-in package called?

- A. reg
- B. regex
- C. re**
- D. regx

2. Which function returns a list containing all matches?

- A. findall**
- B. search
- C. split
- D. find

3. Which character stand for Starts with in regex?

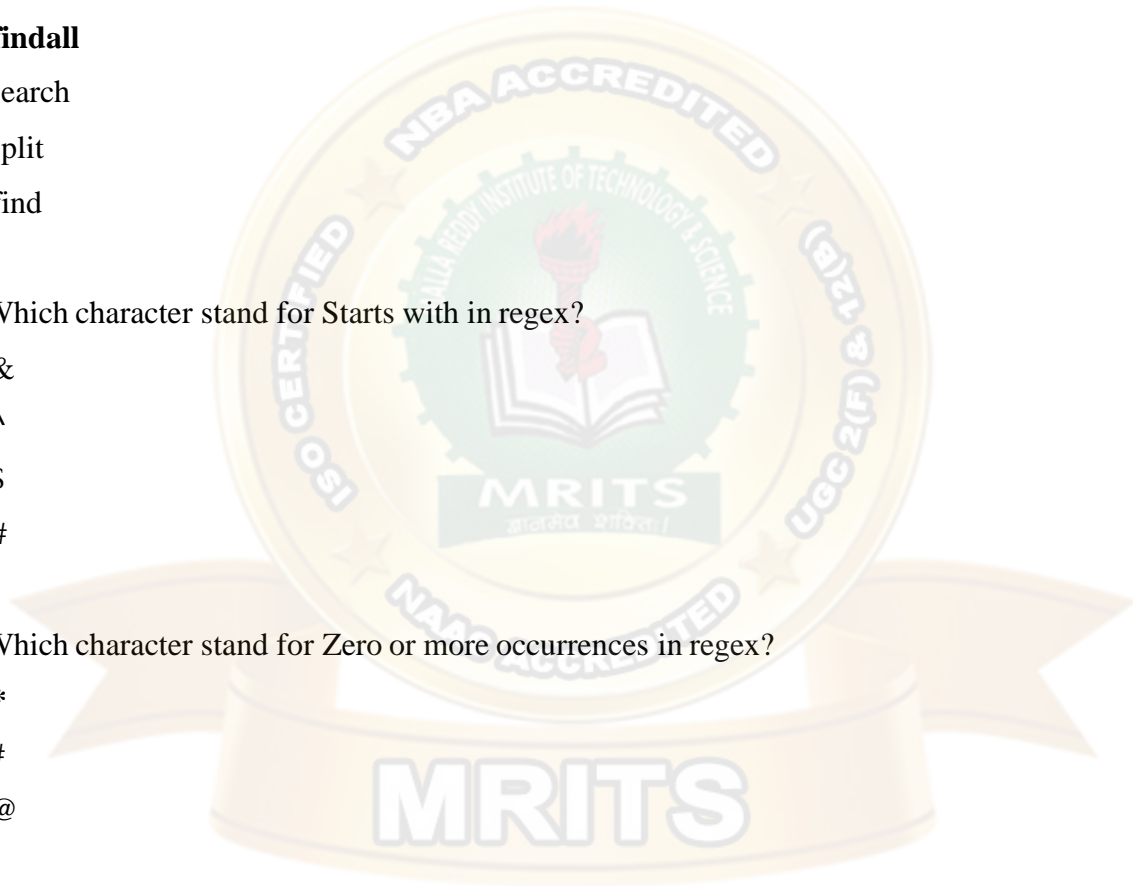
- A. &
- B. ^
- C. \$
- D. #

4. Which character stand for Zero or more occurrences in regex?

- A. *
- B. #
- C. @
- D. |

5. In Regex, s stands for?

- A. Returns a match where the string DOES NOT contain digits
- B. Returns a match where the string DOES NOT contain a white space character
- C. Returns a match where the string contains a white space character**
- D. Returns a match if the specified characters are at the end of the string



6. In Regex, [a-n] stands for?

A. Returns a match for any digit between 0 and 9

B. Returns a match for any lower case character, alphabetically between a and n

C. Returns a match for any two-digit numbers from 00 and 59

D. Returns a match for any character EXCEPT a, r, and n

7. The expression a{5} will match_____characters with the previous regular expression.

A. 5 or less

B. exactly 5

C. 5 or more

D. exactly 4

8. Which of the following functions clears the regular expression cache?

A. re.sub()

B. re.pos()

C. re.purge()

D. re.subn()

9. _____makes it possible for two or more activities to execute in parallel on a single processor.

a) Multithreading

b) Threading

c) SingleThreading

d) Both Multithreading and SingleThreading

10. In_____an object of type Thread in the namespace System.Threading represents and controls one thread.

a) .PY

b) .SAP

c) .NET

d) .EXE

11. The method will be executed once the thread's _____ method is called.

- a) EventBegin
- b) EventStart
- c) Begin
- d) Start**

12. Command to make thread sleep?

- a) Thread.Sleep**
- b) Thread_Sleep
- c) ThreadSleep
- d) Thread_sleep

13. An instance of class Buffer provides a threadsafe way of communication between _____

- a) Actors
- b) Objects**
- c) Locking
- d) Buffer

14. _____ method puts zero into the buffer.

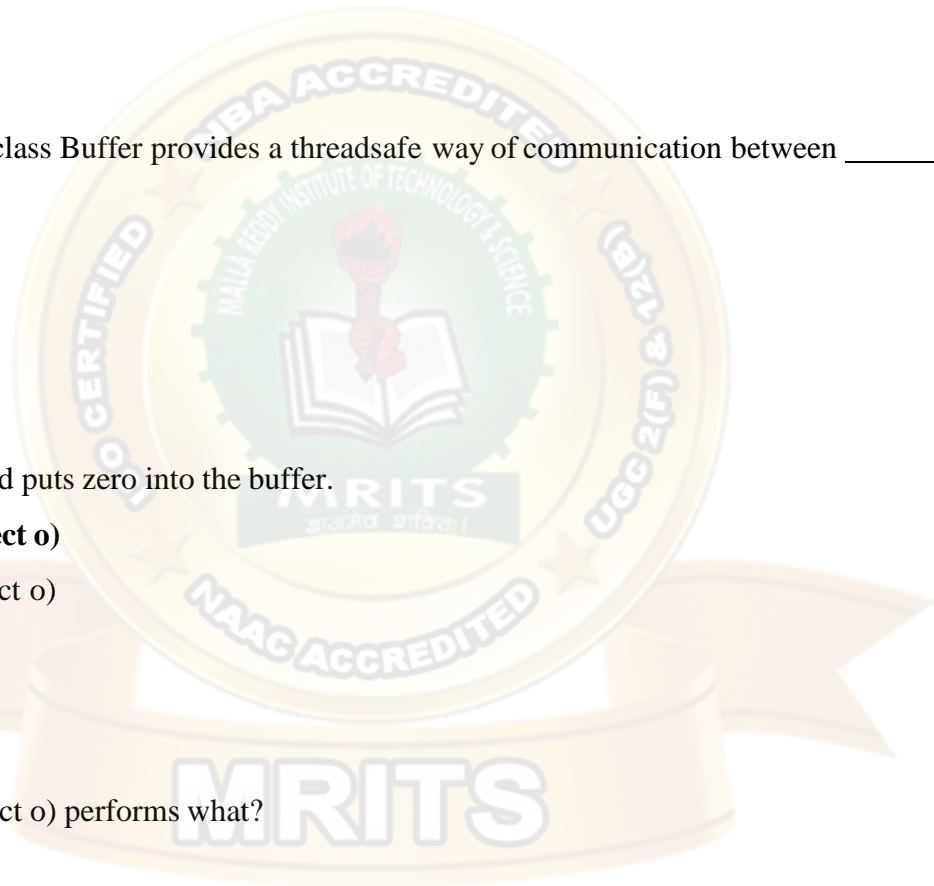
- a) HandlePut(object o)**
- b) HandletGet(object o)
- c) HandletGet()
- d) HandletPut()**

15. HandlePut(object o) performs what?

- a) Fixing values
- b) Locking
- c) Changing values
- d) Unlocking**

16. In HandlePut(object o), o represents?

- a) Null**
- b) Zero
- c) Empty
- d) Origin



17. What is HandleGet() method function?

- a) Current buffer state, with changing
- b) Current buffer state, without changing**
- c) Previous buffer state, with changing
- d) Previous buffer state, without changing

18. What is the result for HandleGet()?

- a) Null**
- b) Zero
- c) Empty
- d) Origin

19. Multithreading is a mechanism for splitting up a program into several parallel activities called _____

- a) Methods
- b) Objects
- c) Classes**
- d) Threads

20. Each thread is a single stream of execution.

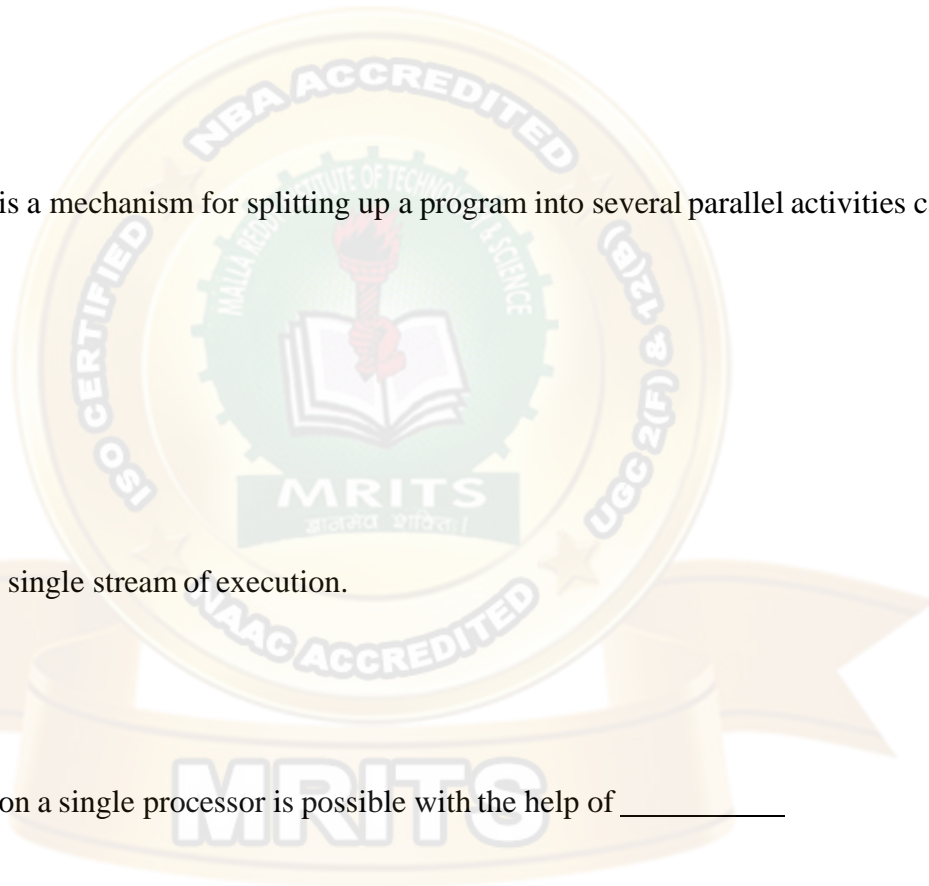
- a) False**
- b) True

21. Multithreading on a single processor is possible with the help of _____

- a) Threader
- b) Scheduler**
- c) Method
- d) Divider

22. Scheduler switch threads in _____

- a) Multilevel queue scheduling
- b) Priority Scheduling
- c) Round robin fashion**
- d) Multilevel feedback queue scheduling



23. What is the switching speed?

- a) 60 times per second
- b) 50 times per second**
- c) 55 times per second
- d) 66 times per second



UNIT-IV

1. In Python context, what does GUI stand for?

- a. General User Interface
- b. Golfing Union of Ireland
- c. Graphical User Interface**
- d. Graphical Unit Interface

2. What's the correct Python code for a script that displays the above message box using EasyGUI?

**a. import easygui as gui
gui.msgbox("Hello, EasyGUI!",
 "This is a message box",
 "Hi there")**

b. easygui.msgbox("Hello, EasyGUI!",
 "This is a message box",
 "Hi there")

c. import easygui as gui
gui.message("Hello, EasyGUI!",
 "This is a message box",
 "Hi there")

d. import easygui as gui
gui.msgbox("Hello, EasyGUI!",
 "Hi there",
 "This is a message box")

3. Which of the following are valid Tkinter widgets? Select all that apply.

- a. ColorPicker
- b. Button**
- c. WebBrowser
- d. Label**

4. Which of the following geometry managers are available in Tkinter?

- a. .table()
- b. .flex()
- c. .grid()**
- d. .pack()**

5. Config() in python are used for

- a. Destroy the widget
- b. Place the widget
- c. Configure the widget
- d. Change the properties of the widget**

6. Essential thing to create a window screen using Tkinter python?

- a. Call tk() function**
- b. Create a button
- c. Define geometry
- d. All of the above

7. fg in tkinter widget stands for?

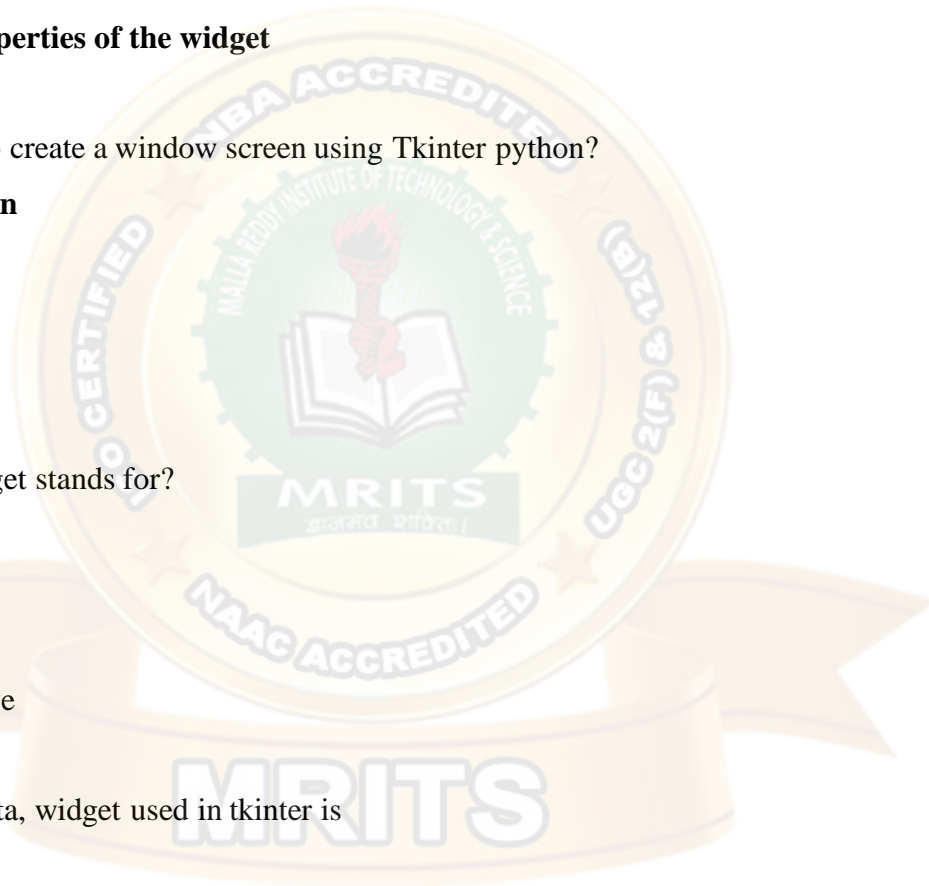
- a. foreground**
- b. background
- c. forgap
- d. None of the above

8. For user entry data, widget used in tkinter is

- a. Entry
- b. Text**
- c. Both
- d. None of the above

9. bg in tkinter widget is used for?

- a. To change the direction of widget
- b. To change the width of widget
- c. To change the background of widget**
- d. To change the color of widget



10. Keyword used to import the tkinter in program is?

- a. call
- b. from
- c. import**
- d. All of the above

11. How to install tkinter in system?

- a. pip install python
- b. pip install tkinter**
- c. tkinter install
- d. None of the above

12. Screen inside another screen is possible by creating

- a. Another Window
- b. Frames**
- c. Buttons
- d. Labels

13. title() is used for_____

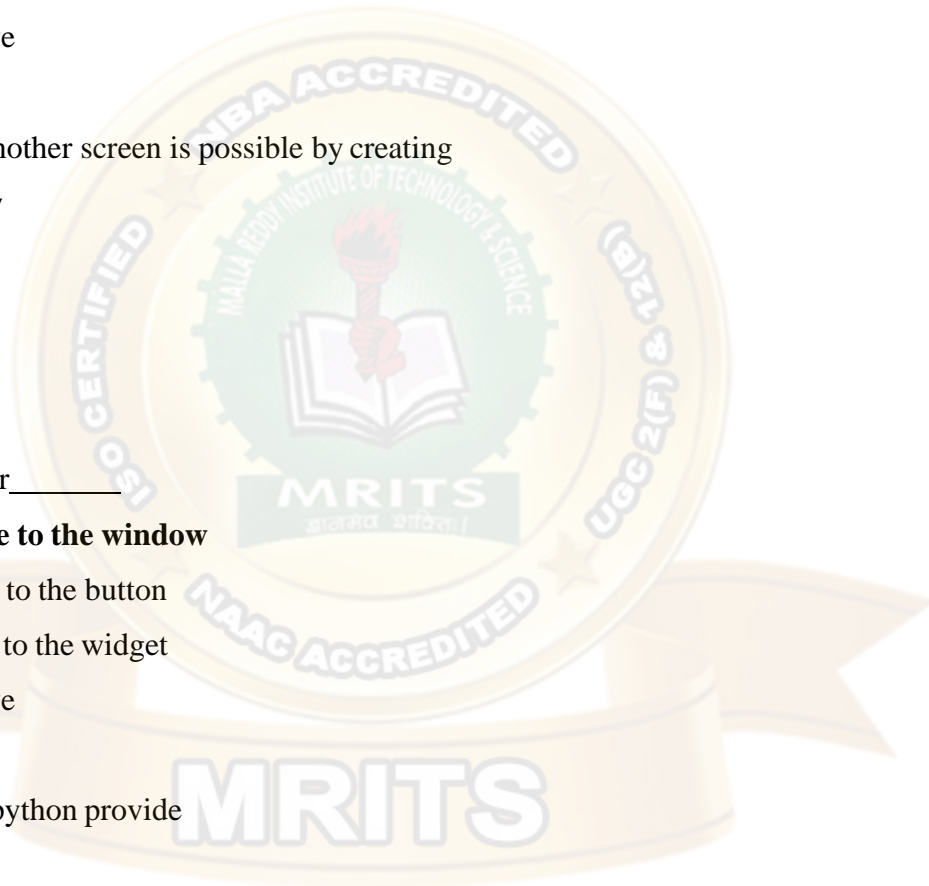
- a. giving title name to the window**
- b. giving title name to the button
- c. giving title name to the widget
- d. None of the above

14. Tkinter tool in python provide

- a. Database
- b. OS commands
- c. GUI**
- d. All of the above

15. To delete any widget from the screen which function we use?

- a. stop()
- b. delete()
- c. destroy()**
- d. break()



16. To hold the screen what we use?

- a. **mainloop() function**
- b. pause()function
- c. stop()function
- d. None of the above

17. What is the correct syntax of destroy in tkinter?

- a. destroy(object)
- b. **object.destroy()**
- c. object(destroy)
- d. delete(object)

18. Which of the following we can able to delete using destroy() function?

- a. Button
- b. Label
- c. Frame
- d. **All of the above**

19. What protocol can be used to retrieve web pages using python?

- A. urllib
- B. bs4
- C. **HTTP**
- D. GET

20. What provides two way communication between two different programs in a network.

- A. **socket**
- B. port
- C. http
- D. protocol

21. What is a python library that can be used to send and receive data over HTTP?

- A. http
- B. **urllib**
- C. port
- D. header

22. What is the process by which search engines retrieve webpages and build a search index called?

- A. scrape
- B. parse
- C. BeautifulSoup
- D. spider**

23. What does the following regex match?

`http[s]?://.+?`

- A. Exact match to 'http[s]?://.+?'
- B. 'http://' or 'http[s]://' followed by one or more character
- C. 'http://' or 'https://' followed by one or more characters.**
- D. 'https://' followed by one or more characters.



UNIT-V

1. What does SQL stand for?

- a. Stylish Query Lingo
- b. Server Query Language
- c. Super Quantum Logic
- d. Structured Query Language**

2. What is the name of the SQL database that comes distributed with Python?

- a. SQLite**
- b. MySQL
- c. PostgreSQL
- d. PySQL

3. Which of the following code snippets creates and connects to a new SQLite Database?

a. import sqlite3

connection = sqlite3.connect("test_database.db")

b. import sql

connection = sql.connect("test_database.db")

c. from sqlite3 import Connection

connection = Connection("test_database.db")

d. import sqlite3

connection = sqlite3.create("test_database.db")

4. Which of the following are valid Cursor methods used to execute SQL statements and retrieve query results? Select all that apply.

a. Cursor.fetchall()

b. Cursor.execute()

c. Cursor.fetchmany()

d. Cursor.run()

e. Cursor.fetchone()

5. Suppose you have a SQLite database called “people.db” with the following table named People:

ID	FirstName	LastName
1	Grace	Hopper
2	Mary	Keller
3	Ada	Lovelace

What is the type of the results variable in the following code snippet?

```
import sqlite3
connection = sqlite3.connection("people.db")
cursor = connection.cursor()
cursor.execute("SELECT * FROM People")
results = cursor.fetchall()
```

- a. QuerySet
- b. tuple
- c. list**
- d. dict



JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B. Tech IV Year I Semester Examinations, December - 2019
PYTHON PROGRAMMING
(Common to CSE, IT)

Time: 3 Hours

Max. Marks: 75

Note: This question paper contains two parts A and B.

Part A is compulsory which carries 25 marks. Answer all questions in Part A. Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b as sub questions.

PART – A

(25 Marks)

- 1.a) State any four applications where python is more popular. [2]
- b) List out the main differences between lists and tuples. [3]
- c) What are the uses of File object? [2]
- d) Give a brief description of several Built-in attributes related to File objects. [3]
- e) Summarize the purpose of pipe and dot symbols used for pattern matching. [2]
- f) Explain the basic functionality of match() function. [3]
- g) What is the need of Tkinter module in python? [2]
- h) How to create Label widget in Python? [3]
- i) State the need of persistent storage. [2]
- j) Discuss the SQL commands/statements used for creating, using and dropping a database. [3]

PART – B

(50 Marks)

- 2.a) How to declare and call functions in Python programs? Illustrate with an example script.
 - b) List and explain few most commonly used built-in types in python. [5+5]
- OR
3. Summarize various operators, built-in functions and standard library modules that deals with Python's numeric type. [10]
 4. Explain the following file built-in functions and method with clear syntax, description and illustration:
a) open() b) file() c) seek() d) tell() e)read() [10]
- OR
- 5.a) How does try-except statement work? Demonstrate with an example python code.
 - b) Illustrate the concept of importing module attributes in python scripts. [5+5]
 6. Examine how python supports regular expressions through the 're' module with brief introduction and various built-in methods related to it. [10]
- OR
- 7.a) What is the motivation behind parallelism and state how python achieves parallelism?
 - b) Explain briefly about thread and threading module objects in Python. [3+7]

8. Consider a Python GUI program that produces a window with the following widgets using python code:

a) A button to retrieve the next value in that list(if there is one).This button is displayed if there is no next value in the list

b) A label to display the number of the items being displayed and the total number of items [10]

OR

9. Give an overview and demonstration of building web applications using python's cgi module. [10]

10.a) What is a cursor object? Explain various methods and attributes of cursor object.

b) What do you mean by a constructor? List and describe various constructors used for converting to different data types. [5+5]

OR

11. Describe in detail about Python SQLAlchemy ORM with a case study of Employee role database. [10]

---ooOoo---

