

III SEMESTER

Core Course- XIII-17PCS09 OPEN SOURCE COMPUTING

Credits: 4

Course Objective:

- To understand the basic Concepts of Python

UNIT - I

Python: Introduction – Numbers – Strings – Variables – Lists – Tuples – Dictionaries – Sets – Comparison.

UNIT - II

Code Structures: if, elif, and else – Repeat with while – Iterate with for – Comprehensions – Functions – Generators – Decorators – Namespaces and Scope – Handle Errors with try and except – User Exceptions.

Modules, Packages, and Programs: Standalone Programs – Command-Line Arguments – Modules and the import Statement – The Python Standard Library. **Objects and Classes:** Define a Class with class – Inheritance – Override a Method – Add a Method – Get Help from Parent with super – In self Defense – Get and Set Attribute Values with Properties – Name Mangling for Privacy – Method Types – Duck Typing – Special Methods – Composition

UNIT-III

Data Types: Text Strings – Binary Data. **Storing and Retrieving Data:** File Input/Output – Structured Text Files – Structured Binary Files - Relational Databases – NoSQL Data Stores.

UNIT-IV

Web: Web Clients – Web Servers – Web Services and Automation – **Systems:** Files – Directories – Programs and Processes – Calendars and Clocks

UNIT-V

Concurrency: Queues – Processes – Threads – Green Threads and gevent – twisted – Redis. **Networks:** Patterns – The Publish-Subscribe Model – TCP/IP – Sockets – ZeroMQ – Internet Services – Web Services and APIs – Remote Processing – Big Fat Data and MapReduce – Working in the Clouds.

TEXT BOOK

1. Bill Lubanovic, "Introducing Python", O'Reilly, First Edition-Second Release, 2014.

REFERENCE BOOKS

1. Mark Lutz, "Learning Python", O'Reilly, Fifth Edition, 2013.
David M. Beazley, "Python Essential Reference", Developer's Library, Fourth Edition, 2009.

UNIT I

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features

Python's features include –

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

- **Databases** – Python provides interfaces to all major commercial databases.
- **GUI Programming** – Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** – Python provides a better structure and support for large programs than shell scripting.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 = 1
```

```
var2 = 10
```

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
del var1[,var2[,var3[.....,varN]]]]
```

You can delete a single object or multiple objects by using the del statement. For example –

```
del var
```

```
del var_a, var_b
```

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEI	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by $x + yj$, where x and y are the real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator (`[]` and `[:]`) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
str = 'Hello World!'
```

```
print str      # Prints complete string
```

```
print str[0]   # Prints first character of the string
```

```
print str[2:5] # Prints characters starting from 3rd to 5th
print str[2:] # Prints string starting from 3rd character
print str * 2 # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result –

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5 ];
```

```
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5, 6, 7 ];
```

```
print "list1[0]: ", list1[0]
```

```
print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result –

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

Updating 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. For example –

```
list = ['physics', 'chemistry', 1997, 2000];
```

```
print "Value available at index 2 : "
```

```
print list[2]
```

```
list[2] = 2001;
```

```
print "New value available at index 2 : "
```

```
print list[2]
```

Note – append() method is discussed in subsequent section.

When the above code is executed, it produces the following result –

Value available at index 2 :

1997

New value available at index 2 :

2001

Delete List Elements

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. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
print list1
```

```
del list1[2];
```

```
print "After deleting value at index 2 : "
```

```
print list1
```

When the above code is executed, it produces following result –

```
['physics', 'chemistry', 1997, 2000]
```

After deleting value at index 2 :

```
['physics', 'chemistry', 2000]
```

Note – remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated. A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5 );
```

```
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5, 6, 7 );
```

```
print "tup1[0]: ", tup1[0];
```

```
print "tup2[1:5]: ", tup2[1:5];
```

When the above code is executed, it produces the following result –

```
tup1[0]: physics
```

```
tup2[1:5]: [2, 3, 4, 5]
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
# Following action is not valid for tuples
# tup1[0] = 100;
```

So let's create a new tuple as follows

```
tup3 = tup1 + tup2;
print tup3;
```

When the above code is executed, it produces the following result –

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

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. For example –

```
tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more –

```
('physics', 'chemistry', 1997, 2000)
```

After deleting tup :

Traceback (most recent call last):

```
File "test.py", line 9, in <module>
```

```
    print tup;
```

NameError: name 'tup' is not defined

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations used on strings in the prior chapter –

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!') * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type,

but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
print "dict['Name']: ", dict['Name']  
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result –

```
dict['Name']: Zara  
dict['Age']: 7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
print "dict['Alice']: ", dict['Alice']
```

When the above code is executed, it produces the following result –

```
dict['Alice']:
```

Traceback (most recent call last):

```
File "test.py", line 4, in <module>  
    print "dict['Alice']: ", dict['Alice'];
```

KeyError: 'Alice'

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
dict['Age'] = 8; # update existing entry  
dict['School'] = "DPS School"; # Add new entry  
  
print "dict['Age']: ", dict['Age']
```

```
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result –

```
dict['Age']: 8
```

```
dict['School']: DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}  
del dict['Name']; # remove entry with key 'Name'  
dict.clear();    # remove all entries in dict  
del dict ;      # delete entire dictionary
```

```
print "dict['Age']: ", dict['Age']  
print "dict['School']: ", dict['School']
```

This produce the following result –

```
This is one
```

```
This is two
```

```
{'dept': 'sales', 'code': 6734, 'name': 'john'}
```

```
['dept', 'code', 'name']
```

```
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
for d in Days:
    print(d)
```

When the above code is executed, it produces the following result.

```
Wed
Sun
Fri
Tue
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
```

```
Days.add("Sun")
```

```
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using `discard()` method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
```

```
Days.discard("Sun")
```

```
print(Days)
```

When the above code is executed, it produces the following result.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
```

```
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
```

```
AllDays = DaysA|DaysB
```

```
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
```

```
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
```

```
AllDays = DaysA & DaysB
```

```
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)
```

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

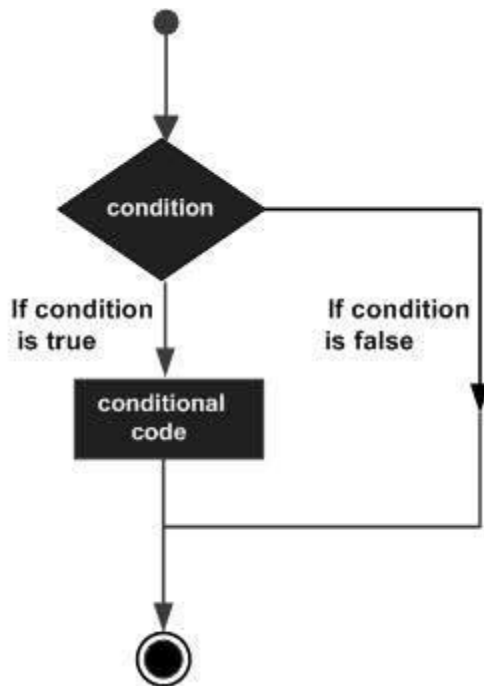
UNIT II

Decision making

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages –



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	<u>if statements</u> An if statement consists of a boolean expression followed by one or more statements.

2	<u>if...else statements</u> An if statement can be followed by an optional else statement , which executes when the boolean expression is FALSE.
3	<u>nested if statements</u> You can use one if or else if statement inside another if or else if statement(s).

Let us go through each decision making briefly –

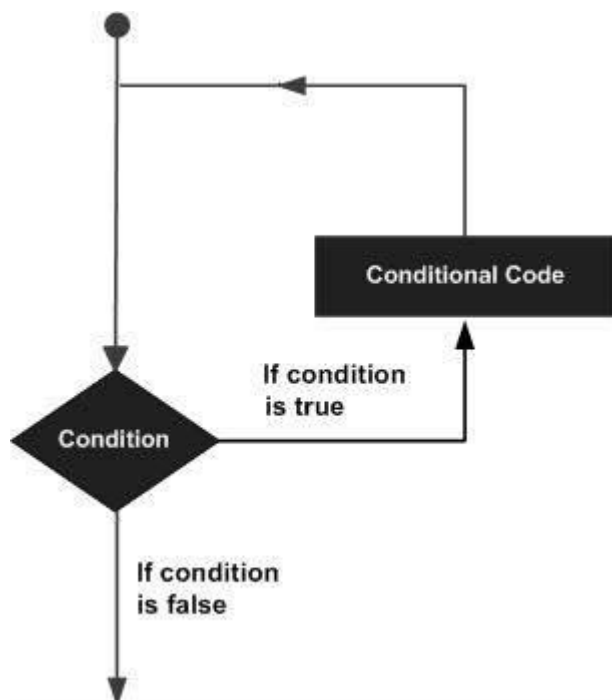
Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

```
var = 100
if ( var == 100 ) : print "Value of expression is 100"
print "Good bye!"
```

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –



Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

Comprehensions in Python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Python supports the following 4 types of comprehensions:

- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions
- Generator Comprehensions

Function

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses `(())`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.

- The code block within every function starts with a colon (:) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Syntax

```
def functionname( parameters ):
```

```
    "function_docstring"
```

```
    function_suite
```

```
    return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print str
```

```
    return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call `printme()` function –

```
# Function definition is here
```

```
def printme( str ):
```

```
    "This prints a passed string into this function"
```

```
    print str
```

```
    return;
```

```
# Now you can call printme function
```

```
printme("I'm first call to user defined function!")
```

```
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result –

```
I'm first call to user defined function!
```

```
Again second call to the same function
```

Generator Comprehensions:

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient. Let's look at the following example to understand generator comprehension:

```
filter_none
```

```
edit
```

```
play_arrow
```

```
brightness_4
```

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
```

```
output_gen = (var for var in input_list if var % 2 == 0)
```

```
print("Output values using generator comprehensions:", end = ' ')
```

```
for var in output_gen:
```

```
    print(var, end = ' ')
```

Decorator

A decorator is a function that takes another function as an argument, does some actions, and then returns the argument based on the actions performed. Since functions are [first-class](#) object in Python, they can be passed as arguments to another functions. Hence we can say that a decorator is a [callable](#) that accepts and returns a callable. Below code shows a simple decorator that add additional notes to the my_function [docstring](#):

```
filter_none
```

```
edit
```

```
play_arrow
```

```
brightness_4
```

```
def decorated_docstring(function):
```

```
function.__doc__ += '\n Hi George, I'm a simple Decorator.'  
return function
```

```
def my_function(string1):  
    """Return the string."""  
    return string1
```

```
def main():  
    myFunc = decorated_docstring(my_function)  
    myFunc('Hai')  
    help(myFunc)
```

```
if __name__ == "__main__":  
    main()
```

Output:

Help on function my_function in module __main__:

```
my_function(string1)  
    Return the string.  
    Hi George, Im a simple Decorator.
```

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an `UnboundLocalError` is the result. Uncommenting the global statement fixes the problem.

```
Money = 2000
```

```
def AddMoney():
```

```
    # Uncomment the following line to fix the code:
```

```
    # global Money
```

```
    Money = Money + 1
```

```
print Money
```

```
AddMoney()
```

```
print Money
```

The `dir()` Function

The `dir()` built-in function **returns** a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example –

```
# Import built-in module math
```

```
import math
```

Exception Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them –

- Exception Handling – This would be covered in this tutorial. Here is a list standard Exceptions available in Python: [Standard Exceptions](#).
- Assertions – This would be covered in [Assertions in Python](#) tutorial.

List of Standard Exceptions –

Sr.No.	Exception Name & Description
1	Exception Base class for all exceptions
2	StopIteration Raised when the next() method of an iterator does not point to any object.
3	SystemExit Raised by the sys.exit() function.
4	StandardError Base class for all built-in exceptions except StopIteration and SystemExit.
5	ArithmeticError Base class for all errors that occur for numeric calculation.
6	OverflowError Raised when a calculation exceeds maximum limit for a numeric type.
7	FloatingPointError Raised when a floating point calculation fails.
8	ZeroDivisionError Raised when division or modulo by zero takes place for all numeric types.

9	<p>AssertionError</p> <p>Raised in case of failure of the Assert statement.</p>
10	<p>AttributeError</p> <p>Raised in case of failure of attribute reference or assignment.</p>
11	<p>EOFError</p> <p>Raised when there is no input from either the raw_input() or input() function and the end of file is reached.</p>
12	<p>ImportError</p> <p>Raised when an import statement fails.</p>
13	<p>KeyboardInterrupt</p> <p>Raised when the user interrupts program execution, usually by pressing Ctrl+c.</p>
14	<p>LookupError</p> <p>Base class for all lookup errors.</p>
15	<p>IndexError</p> <p>Raised when an index is not found in a sequence.</p>
16	<p>KeyError</p> <p>Raised when the specified key is not found in the dictionary.</p>
17	<p>NameError</p> <p>Raised when an identifier is not found in the local or global namespace.</p>
18	<p>UnboundLocalError</p> <p>Raised when trying to access a local variable in a function or method but no value has been assigned to it.</p>
19	<p>EnvironmentError</p> <p>Base class for all exceptions that occur outside the Python environment.</p>

20	<p>IOError</p> <p>Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.</p>
21	<p>IOError</p> <p>Raised for operating system-related errors.</p>
22	<p>SyntaxError</p> <p>Raised when there is an error in Python syntax.</p>
23	<p>IndentationError</p> <p>Raised when indentation is not specified properly.</p>
24	<p>SystemError</p> <p>Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.</p>
25	<p>SystemExit</p> <p>Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.</p>
26	<p>TypeError</p> <p>Raised when an operation or function is attempted that is invalid for the specified data type.</p>
27	<p>ValueError</p> <p>Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.</p>
28	<p>RuntimeError</p> <p>Raised when a generated error does not fall into any category.</p>
29	<p>NotImplementedError</p> <p>Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.</p>

Module

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, *support.py*

```
def print_func( par ):
    print "Hello : ", par
    return
```

The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax –

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module *support.py*, you need to put the following command at the top of the script –

```
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("Zara")
```

When the above code is executed, it produces the following result –

```
Hello : Zara
```

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.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function *fibonacci* from the module *fib*, use the following statement –

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module.

The *from...import ** Statement

It is also possible to import all names from a module into the current namespace by using the following import statement –

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code
def Pots():

```
    print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above –

- *Phone/Isdn.py* file having function `Isdn()`
- *Phone/G3.py* file having function `G3()`

Now, create one more file `__init__.py` in *Phone* directory –

- *Phone/__init__.py*

To make all of your functions available when you've imported *Phone*, you need to put explicit import statements in `__init__.py` as follows –

```
from Pots import Pots
```

```
from Isdn import Isdn
```

```
from G3 import G3
```

After you add these lines to `__init__.py`, you have all of these classes available when you import the *Phone* package.

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

```
# Now import your Phone Package.
```

```
import Phone
```

Phone.Pots()

Phone.Isdn()

Phone.G3()

When the above code is executed, it produces the following result –

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

Classes and Objects

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed –

Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

- **Instantiation** – The creation of an instance of a class.
- **Method** – A special kind of function that is defined in a class definition.
- **Object** – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows –

```
class ClassName:
```

```
'Optional class documentation string'
```

```
class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class –

```
class Employee:
```

```
'Common base class for all employees'
```

```
empCount = 0
```

```
def __init__(self, name, salary):
```

```
    self.name = name
```

```
    self.salary = salary
```

```
    Employee.empCount += 1
```

```
def displayCount(self):
```

```
    print "Total Employee %d" % Employee.empCount
```

```
def displayEmployee(self):
```

```
    print "Name : ", self.name, ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method `__init__()` is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its `__init__` method accepts.

"This would create first object of Employee class"

```
emp1 = Employee("Zara", 2000)
```

"This would create second object of Employee class"

```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows –

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

Now, putting all the concepts together –

```
class Employee:
```

```
    'Common base class for all employees'
```

```
    empCount = 0
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
        Employee.empCount += 1
```

```
    def displayCount(self):
```

```
print "Total Employee %d" % Employee.empCount
```

```
def displayEmployee(self):
```

```
    print "Name : ", self.name, ", Salary: ", self.salary
```

"This would create first object of Employee class"

```
emp1 = Employee("Zara", 2000)
```

"This would create second object of Employee class"

```
emp2 = Employee("Manni", 5000)
```

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

When the above code is executed, it produces the following result –

```
Name : Zara ,Salary: 2000
```

```
Name : Manni ,Salary: 5000
```

```
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.age = 7 # Add an 'age' attribute.
```

```
emp1.age = 8 # Modify 'age' attribute.
```

```
del emp1.age # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj,name)** – to check if an attribute exists or not.
- The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
hasattr(emp1, 'age') # Returns true if 'age' attribute exists
```

```
getattr(emp1, 'age') # Returns value of 'age' attribute
```

```
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
```

```
delattr(emp1, 'age') # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –

- `__dict__` – Dictionary containing the class's namespace.
- `__doc__` – Class documentation string or none, if undefined.
- `__name__` – Class name.
- `__module__` – Module name in which the class is defined. This attribute is "`__main__`" in interactive mode.
- `__bases__` – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes –

```
class Employee:
```

```
    'Common base class for all employees'
```

```
    empCount = 0
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
        Employee.empCount += 1
```

```
    def displayCount(self):
```

```
        print "Total Employee %d" % Employee.empCount
```

```
    def displayEmployee(self):
```

```
        print "Name : ", self.name, ", Salary: ", self.salary
```

```
print "Employee.__doc__:", Employee.__doc__
```

```
print "Employee.__name__:", Employee.__name__
```

```
print "Employee.__module__:", Employee.__module__
```

```
print "Employee.__bases__:", Employee.__bases__
```

```
print "Employee.__dict__:", Employee.__dict__
```

When the above code is executed, it produces the following result –

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40    # Create object <40>
b = a     # Increase ref. count of <40>
c = [b]   # Increase ref. count of <40>

del a     # Decrease ref. count of <40>
b = 100   # Decrease ref. count of <40>
c[0] = -1 # Decrease ref. count of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method `__del__()`, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

Example

This `__del__()` destructor prints the class name of an instance that is about to be destroyed –


```

class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the objects
del pt1
del pt2
del pt3

```

When the above code is executed, it produces following result –

```
3083401324 3083401324 3083401324
```

```
Point destroyed
```

Class Inheritance

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```

class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite

```

UNIT III

Working with Binary Data in Python

Alright, lets get this out of the way! The basics are pretty standard:

1. There are 8 bits in a byte
2. Bits either consist of a 0 or a 1
3. A byte can be interpreted in different ways, like binary octal or hexadecimal

Examples:

Input : 10011011

Output :

1001 1011 ---- 9B (in hex)

1001 1011 ---- 155 (in decimal)

1001 1011 ---- 233 (in octal)

This clearly shows a string of bits can be interpreted differently in different ways. We often use the hex representation of a byte instead of the binary one because it is shorter to write, this is just a representation and not an interpretation.

Encoding

Now that we know what a byte is and what it looks like, let us see how it is interpreted, mainly in *strings*. Character Encodings are a way to assign values to bytes or sets of bytes that represent a certain character in that scheme. Some encodings are ASCII(probably the oldest), Latin, and UTF-8(most widely used as of today. In a sense encodings are a way for computers to represent, send and interpret human readable characters. This means that a sentence in one encoding might become completely incomprehensible in another encoding.

Python provides the facility to read, write, and create files. The file can be two types - normal text and binary.

- **Text Files** - This type of file consists of the normal characters, terminated by the special character This special character is called EOL (End of Line). In Python, the new line ('\n') is used by default.
- **Binary Files** - In this file format, the data is stored in the binary format (1 or 0). The binary file doesn't have any terminator for a newline.

Here, we will learn to read the text file in Python.

[Python](#) takes the three required steps to read or write a text file.

- Open a file
- Read or Write file

- Close file

Reading a Text File

Python provides a [built-in function](#) `open()` to open a file. It takes mainly two arguments the **filename** and **mode**. It returns the file object, which is also called a handle. It can be used to perform various operations to the file.

1. `fs = open('example.txt', 'r')` # The first argument is the file name, and the second #argument is a mode of the file. Here r means read mode.

We can specify the mode of the file while opening a file. The mode of file can be read **r**, write **w**, and append **a**.

We will open the text file using the `open()` function.

Python provides the various function to read the file, but we will use the most common `read()` function. It takes an argument called size, which is nothing but a given number of characters to be read from the file. If the size is not specified, then it will read the entire file.

Example -

1. `fs = open(r"C:\Users\DEVANSH SHARMA\Desktop\example.txt",'r')`
2. `# It will read the 4 characters from the text file`
3. `con = fs.read(4)`
4. `# It will read the 10 characters from the text file`
5. `con1 = fs.read(10)`
6. `# It will read the entire file`
7. `con2 = fs.read()`
8. `print(con)`
9. `fs.close()` # It will read the entire file

Output:

This

is line 1

This is line 2

This is line 3

This is line 4

Explanation

In the above code, we can see the `read()` function read the character according to its given **size** from the file. The `con1` variable read the **next 10 characters** from the last `read()`

function. In the last line, we closed the file after performing the read operation using the `close()` function.

DATABASE

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as –

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

Here is the list of available Python database interfaces: [Python Database Interfaces and APIs](#). You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following –

- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and stored procedures.
- Closing the connection

We would learn all the concepts using MySQL, so let us talk about MySQLdb module.

What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

How do I Install MySQLdb?

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script and execute it –

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

```
import MySQLdb
```

If it produces the following result, then it means MySQLdb module is not installed –

Traceback (most recent call last):

```
File "test.py", line 3, in <module>
```

```
    import MySQLdb
```

ImportError: No module named MySQLdb

To install MySQLdb module, use the following command –

For Ubuntu, use the following command -

```
$ sudo apt-get install python-pip python-dev libmysqlclient-dev
```

For Fedora, use the following command -

```
$ sudo dnf install python python-devel mysql-devel redhat-rpm-config gcc
```

For Python command prompt, use the following command -

```
pip install MySQL-python
```

Note – Make sure you have root privilege to install above module.

Database Connection

Before connecting to a MySQL database, make sure of the followings –

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module MySQLdb is installed properly on your machine.
- You have gone through MySQL tutorial to understand [MySQL Basics](#).

Example

Following is the example of connecting with MySQL database "TESTDB"

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

```
import MySQLdb
```

```
# Open database connection
```

```
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
```

```
# prepare a cursor object using cursor() method
```

```
cursor = db.cursor()
```

```
# execute SQL query using execute() method.
```

```
cursor.execute("SELECT VERSION()")
```

```
# Fetch a single row using fetchone() method.
```

```
data = cursor.fetchone()
```

```
print "Database version : %s " % data
```

```
# disconnect from server
```

```
db.close()
```

While running this script, it is producing the following result in my Linux machine.

```
Database version : 5.0.45
```

If a connection is established with the datasource, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that database connection is closed and resources are released.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

Example

Let us create Database table EMPLOYEE –

```
import MySQLdb
```

```
# Open database connection
```

```
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
```

```
# prepare a cursor object using cursor() method
```

```
cursor = db.cursor()
```

```
# Drop table if it already exist using execute() method.  
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")
```

```
# Create table as per requirement
```

```
sql = """CREATE TABLE EMPLOYEE (  
    FIRST_NAME CHAR(20) NOT NULL,  
    LAST_NAME CHAR(20),  
    AGE INT,  
    SEX CHAR(1),  
    INCOME FLOAT )"""
```

```
cursor.execute(sql)
```

```
# disconnect from server
```

```
db.close()
```

```
INSERT Operation
```

It is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record into *EMPLOYEE* table –

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

```
import MySQLdb
```

```
# Open database connection
```

```
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
```

```
# prepare a cursor object using cursor() method
```

```
cursor = db.cursor()
```

Prepare SQL query to INSERT a record into the database.

```
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,  
    LAST_NAME, AGE, SEX, INCOME)  
    VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
```

try:

```
# Execute the SQL command  
cursor.execute(sql)  
  
# Commit your changes in the database  
db.commit()
```

except:

```
# Rollback in case there is any error  
db.rollback()
```

disconnect from server

```
db.close()
```

Above example can be written as follows to create SQL queries dynamically –

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

```
import MySQLdb
```

Open database connection

```
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
```

prepare a cursor object using *cursor()* method

```
cursor = db.cursor()
```

Prepare SQL query to INSERT a record into the database.

```
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \  
    LAST_NAME, AGE, SEX, INCOME) \  
    VALUES ('%s', '%s', '%d', '%c', '%d' )" % \  
    \
```



```
('Mac', 'Mohan', 20, 'M', 2000)
```

```
try:
```

```
# Execute the SQL command
cursor.execute(sql)
# Commit your changes in the database
db.commit()
```

```
except:
```

```
# Rollback in case there is any error
db.rollback()
```

```
# disconnect from server
```

```
db.close()
```

Example

```
user_id = "test123"
```

```
password = "password"
```

```
con.execute('insert into Login values("%s", "%s")' % \
            (user_id, password))
```

```
.....
```

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch single record or **fetchall()** method to fetch multiple values from a database table.

- **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall()** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **rowcount** – This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 –

```

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

sql = "SELECT * FROM EMPLOYEE \
      WHERE INCOME > '%d'" % (1000)
try:
    # Execute the SQL command
    cursor.execute(sql)
    # Fetch all the rows in a list of lists.
    results = cursor.fetchall()
    for row in results:
        fname = row[0]
        lname = row[1]
        age = row[2]
        sex = row[3]
        income = row[4]
        # Now print fetched result
        print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
            (fname, lname, age, sex, income )
except:
    print "Error: unable to fetch data"

# disconnect from server
db.close()

```

This will produce the following result –

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as 'M'. Here, we increase AGE of all the males by one year.

Example

```
import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
      WHERE SEX = '%c'" % ('M')

try:
    # Execute the SQL command
    cursor.execute(sql)

    # Commit your changes in the database
    db.commit()

except:
    # Rollback in case there is any error
    db.rollback()

# disconnect from server
db.close()

DELETE Operation
```

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 –

Example

```
import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    # Execute the SQL command
    cursor.execute(sql)
    # Commit your changes in the database
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()

# disconnect from server
db.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency. Transactions have the following four properties –

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.

- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is again similar example –

Prepare SQL query to DELETE required records

```
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
```

try:

```
# Execute the SQL command
```

```
cursor.execute(sql)
```

```
# Commit your changes in the database
```

```
db.commit()
```

except:

```
# Rollback in case there is any error
```

```
db.rollback()
```

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call **commit** method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use **rollback()** method.

Here is a simple example to call **rollback()** method.

```
db.rollback()
```

Disconnecting Database

To disconnect Database connection, use **close()** method.

```
db.close()
```

the basic I/O functions available in Python. For more functions, please refer to standard Python documentation.

Printing to the Screen

The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows –

```
print "Python is really a great language,", "isn't it?"
```

This produces the following result on your standard screen –

```
Python is really a great language, isn't it?
```

Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are –

```
raw_input
```

```
input
```

The raw_input Function

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).

```
str = raw_input("Enter your input: ")
```

```
print "Received input is : ", str
```

This prompts you to enter any string and it would display same string on the screen. When I typed "Hello Python!", its output is like this –

```
Enter your input: Hello Python
```

```
Received input is : Hello Python
```

The input Function

The `input([prompt])` function is equivalent to `raw_input`, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
str = input("Enter your input: ")  
print "Received input is : ", str
```

This would produce the following result against the entered input –

```
Enter your input: [x*5 for x in range(2,10,2)]  
Received input is : [10, 20, 30, 40]
```

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

The open Function

Before you can read or write a file, you have to open it using Python's built-in `open()` function. This function creates a file object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details –

`file_name` – The `file_name` argument is a string value that contains the name of the file that you want to access.

`access_mode` – The `access_mode` determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

Here is a list of the different modes of opening a file –

Sr.No. Modes & Description

1
r

Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.

2

rb

Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.

3

r+

Opens a file for both reading and writing. The file pointer placed at the beginning of the file.

4

rb+

Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.

5

w

Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

6

wb

Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

7

w+

Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

8

wb+

Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

9

a

Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

10

ab

Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

11

a+

Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

12

ab+

Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes

Once a file is opened and you have one file object, you can get various information related to that file.

Here is a list of all attributes related to file object –

Sr.No. Attribute & Description

1

file.closed

Returns true if file is closed, false otherwise.

2

file.mode

Returns access mode with which file was opened.

3

file.name

Returns name of the file.

4

file.softspace

Returns false if space explicitly required with print, true otherwise.

Example

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

This produces the following result –

Name of the file: foo.txt

Closed or not : False

Opening mode : wb

Softspace flag : 0

The close() Method

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close()
```

Structured Text Files

With simple text files, the only level of organization is the line. Sometimes, you want more structure than that. You might want to save data for your program to use later, or send data to another program.

There are many formats, and here's how you can distinguish them:

- A separator, or delimiter, character like tab ('\t'), comma (','), or vertical bar (|). This is an example of the comma-separated values (CSV) format.
- '<' and '>' around tags. Examples include XML and HTML.
- Punctuation. An example is JavaScript Object Notation (JSON).
- Indentation. An example is YAML (which depending on the source you use means “YAML Ain't Markup Language;” you'll need to research that one yourself).
- Miscellaneous, such as configuration files for programs.

Each of these structured file formats can be read and written by at least one Python

module.

CSV

Delimited files are often used as an exchange format for spreadsheets and databases. You could read CSV files manually, a line at a time, splitting each line into fields at comma separators, and adding the results to data structures such as lists and dictionaries. But it's better to use the standard `csv` module, because parsing these files can get more complicated than you think.

- Some have alternate delimiters besides a comma: `'|'` and `'\t'` (tab) are common.
- Some have escape sequences. If the delimiter character can occur within a field, the entire field might be surrounded by quote characters or preceded by some escape character.
- Files have different line-ending characters. Unix uses `'\n'`, Microsoft uses `'\r\n'`, and Apple used to use `'\r'` but now uses `'\n'`.
- There can be column names in the first line.

First, we'll see how to read and write a list of rows, each containing a list of columns:

```
>>> import csv
>>> villains = [
... ['Doctor', 'No'],
... ['Rosa', 'Klebb'],
... ['Mister', 'Big'],
... ['Auric', 'Goldfinger'],
Structured Text Files | 181
... ['Ernst', 'Blofeld'],
... ]
>>> with open('villains', 'wt') as fout: # a context manager
...     csvout = csv.writer(fout)
...     csvout.writerows(villains)
```

This creates the file `villains` with these lines:

```
Doctor,No
Rosa,Klebb
Mister,Big
```

Auric,Goldfinger

Ernst,Blofeld

Now, we'll try to read it back in:

```
>>> import csv
>>> with open('villains', 'rt') as fin: # context manager
... cin = csv.reader(fin)
... villains = [row for row in cin] # This uses a list comprehension
...
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
 ['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

Take a moment to think about list comprehensions

We took advantage of the structure created by the reader() function. It obligingly created rows in the cin object that we can extract in a for loop.

Using reader() and writer() with their default options, the columns are separated by commas and the rows by line feeds.

The data can be a list of dictionaries rather than a list of lists. Let's read the villains file again, this time using the new DictReader() function and specifying the column names:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
... cin = csv.DictReader(fin, fieldnames=['first', 'last'])
... villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
 {'last': 'Klebb', 'first': 'Rosa'},
 {'last': 'Big', 'first': 'Mister'},
 {'last': 'Goldfinger', 'first': 'Auric'},
 {'last': 'Blofeld', 'first': 'Ernst'}]
```

Let's rewrite the CSV file by using the new DictWriter() function. We'll also call write

header() to write an initial line of column names to the CSV file:

```
import csv
villains = [
    {'first': 'Doctor', 'last': 'No'},
    {'first': 'Rosa', 'last': 'Klebb'},
    {'first': 'Mister', 'last': 'Big'},
    {'first': 'Auric', 'last': 'Goldfinger'},
    {'first': 'Ernst', 'last': 'Blofeld'},
]
with open('villains', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

That creates a villains file with a header line:

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

Now we'll read it back. By omitting the fieldnames argument in the DictReader() call, we instruct it to use the values in the first line of the file (first,last) as column labels and matching dictionary keys:

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
```

```
{'last': 'Klebb', 'first': 'Rosa'},  
{'last': 'Big', 'first': 'Mister'},  
{'last': 'Goldfinger', 'first': 'Auric'},  
{'last': 'Blofeld', 'first': 'Ernst'}]
```

XML

Delimited files convey only two dimensions: rows (lines) and columns (fields within a line). If you want to exchange data structures among programs, you need a way to encode hierarchies, sequences, sets, and other structures as text.

XML is the most prominent markup format that suits the bill. It uses tags to delimit data, as in this sample menu.xml file:

```
<?xml version="1.0"?>  
<menu>  
  <breakfast hours="7-11">  
    <item price="$6.00">breakfast burritos</item>  
    <item price="$4.00">pancakes</item>
```

Structured Text Files | 183

```
</breakfast>  
  <lunch hours="11-3">  
    <item price="$5.00">hamburger</item>  
  </lunch>  
  <dinner hours="3-10">  
    <item price="8.00">spaghetti</item>  
  </dinner>  
</menu>
```

Following are a few important characteristics of XML:

- Tags begin with a < character. The tags in this sample were menu, breakfast, lunch, dinner, and item.
- Whitespace is ignored.
- Usually a start tag such as <menu> is followed by other content and then a final matching end tag such as </menu>.

- Tags can nest within other tags to any level. In this example, item tags are children of the breakfast, lunch, and dinner tags; they, in turn, are children of menu.
- Optional attributes can occur within the start tag. In this example, price is an attribute of item.
- Tags can contain values. In this example, each item has a value, such as pancakes for the second breakfast item.
- If a tag named thing has no values or children, it can be expressed as the single tag by including a forward slash just before the closing angle bracket, such as <thing/>, rather than a start and end tag, like <thing></thing>.
- The choice of where to put data—attributes, values, child tags—is somewhat arbitrary. For instance, we could have written the last item tag as <item price="\$8.00" food="spaghetti"/>.

XML is often used for data feeds and messages, and has subformats like RSS and Atom.

Some industries have many specialized XML formats, such as the finance field.

XML's über-flexibility has inspired multiple Python libraries that differ in approach and capabilities.

The simplest way to parse XML in Python is by using ElementTree. Here's a little program to parse the menu.xml file and print some tags and attributes:

```
>>> import xml.etree.ElementTree as et
>>> tree = et.ElementTree(file='menu.xml')
>>> root = tree.getroot()
>>> root.tag
'menu'
>>> for child in root:
... print('tag:', child.tag, 'attributes:', child.attrib)
... for grandchild in child:
... print("\ttag:", grandchild.tag, 'attributes:', grandchild.attrib)
...
tag: breakfast attributes: {'hours': '7-11'}
tag: item attributes: {'price': '$6.00'}
```

```
tag: item attributes: {'price': '$4.00'}
tag: lunch attributes: {'hours': '11-3'}
tag: item attributes: {'price': '$5.00'}
tag: dinner attributes: {'hours': '3-10'}
tag: item attributes: {'price': '8.00'}
>>> len(root) # number of menu sections
3
>>> len(root[0]) # number of breakfast items
2
```

For each element in the nested lists, tag is the tag string and attrib is a dictionary of its attributes. ElementTree has many other ways of searching XML-derived data, modifying it, and even writing XML files. The ElementTree documentation has the details.

Other standard Python XML libraries include:

xml.dom

The Document Object Model (DOM), familiar to JavaScript developers, represents Web documents as hierarchical structures. This module loads the entire XML file into memory and lets you access all the pieces equally.

xml.sax

Simple API for XML, or SAX, parses XML on the fly, so it does not have to load everything into memory at once. Therefore, it can be a good choice if you need to process very large streams of XML.

HTML

Enormous amounts of data are saved as Hypertext Markup Language (HTML), the basic document format of the Web. The problem is so much of it doesn't follow the HTML rules, which can make it difficult to parse. Also, much of HTML is intended more to format output than interchange data. Because this chapter is intended to describe fairly well-defined data formats,

JSON

JavaScript Object Notation (JSON) has become a very popular data interchange format, beyond its JavaScript origins. The JSON format is a subset of JavaScript, and often legal

Python syntax as well. Its close fit to Python makes it a good choice for data interchange among programs.

Unlike the variety of XML modules, there's one main JSON module, with the unforgettable name `json`. This program encodes (dumps) data to a JSON string and decodes (loads) a JSON string back to data. In this next example, let's build a Python data structure containing the data from the earlier XML example:

```
>>> menu = \
... {
... "breakfast": {
... "hours": "7-11",
... "items": {
... "breakfast burritos": "$6.00",
... "pancakes": "$4.00"
... }
... },
... "lunch" : {
... "hours": "11-3",
... "items": {
... "hamburger": "$5.00"
... }
... },
... "dinner": {
... "hours": "3-10",
... "items": {
... "spaghetti": "$8.00"
... }
... }
... }
.
```

Next, encode the data structure (`menu`) to a JSON string (`menu_json`) by using `dumps()`:

```
>>> import json
>>> menu_json = json.dumps(menu)
>>> menu_json
'{"dinner": {"items": {"spaghetti": "$8.00"}, "hours": "3-10"},
"lunch": {"items": {"hamburger": "$5.00"}, "hours": "11-3"},
"breakfast": {"items": {"breakfast burritos": "$6.00", "pancakes":
"$4.00"}, "hours": "7-11"}}'
```

And now, let's turn the JSON string `menu_json` back into a Python data structure (`menu2`) by using `loads()`:

```
>>> menu2 = json.loads(menu_json)
>>> menu2
{'breakfast': {'items': {'breakfast burritos': '$6.00', 'pancakes':
'$4.00'}, 'hours': '7-11'}, 'lunch': {'items': {'hamburger': '$5.00'},
'hours': '11-3'}, 'dinner': {'items': {'spaghetti': '$8.00'}, 'hours': '3-10'}}
```

`menu` and `menu2` are both dictionaries with the same keys and values. As always with standard dictionaries, the order in which you get the keys varies.

You might get an exception while trying to encode or decode some objects, including objects such as `datetime` as demonstrated here.

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
>>> json.dumps(now)
```

Traceback (most recent call last):

```
# ... (deleted stack trace to save trees)
```

```
TypeError: datetime.datetime(2013, 2, 22, 3, 49, 27, 483336) is not JSON serializable
```

```
>>>
```

This can happen because the JSON standard does not define date or time types; it expects you to define how to handle them. You could convert the `datetime` to something JSON understands, such as a string or an epoch value (coming in Chapter 10):

```

>>> now_str = str(now)
>>> json.dumps(now_str)
'"2013-02-22 03:49:27.483336"'
>>> from time import mktime
>>> now_epoch = int(mktime(now.timetuple()))
>>> json.dumps(now_epoch)
'1361526567'

```

If the datetime value could occur in the middle of normally converted data types, it might be annoying to make these special conversions. You can modify how JSON is encoded by using inheritance, which is described in “Inheritance” on page 126. Python’s JSON documentation gives an example of this for complex numbers, which also makes JSON play dead. Let’s modify it for datetime:

```

>>> class DTEncoder(json.JSONEncoder):
... def default(self, obj):
... # isinstance() checks the type of obj
... if isinstance(obj, datetime.datetime):
... return int(mktime(obj.timetuple()))
... # else it's something the normal decoder knows:
... return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(now, cls=DTEncoder)
'1361526567'

```

The new class DTEncoder is a subclass, or child class, of JSONEncoder. We only need to override its default() method to add datetime handling. Inheritance ensures that everything else will be handled by the parent class.

The isinstance() function checks whether the object obj is of the class datetime.datetime. Because everything in Python is an object, isinstance() works everywhere:

Relational Databases

Relational databases are only about 40 years old but are ubiquitous in the computing world. You’ll almost certainly have to deal with them at one time or another. When you

do, you'll appreciate what they provide:

- Access to data by multiple simultaneous users
- Protection from corruption by those users
- Efficient methods to store and retrieve the data
- Data defined by schemas and limited by constraints
- Joins to find relationships across diverse types of data
- A declarative (rather than imperative) query language: SQL (Structured Query Language)

SQL

SQL is not an API or a protocol, but a declarative language: you say what you want rather than how to do it. It's the universal language of relational databases. SQL queries are text strings, that a client sends to the database server, which figures out what to do with them. There have been various SQL standard definitions, but all database vendors have added their own tweaks and extensions, resulting in many SQL dialects. If you store your data in a relational database, SQL gives you some portability. Still, dialect and operational differences can make it difficult to move your data to another type of database.

There are two main categories of SQL statements:

DDL (data definition language)

Handles creation, deletion, constraints, and permissions for tables, databases, and users

DML (data manipulation language)

Handles data insertions, selects, updates, and deletions

SQLite

SQLite is a good, light, open source relational database. It's implemented as a standard Python library, and stores databases in normal files. These files are portable across machines and operating systems, making SQLite a very portable solution for simple relational database applications. It isn't as full-featured as MySQL or PostgreSQL, but it does support SQL, and it manages multiple simultaneous users. Web browsers, smart phones, and other applications use SQLite as an embedded database.

You begin with a `connect()` to the local SQLite database file that you want to use or create. This file is the equivalent of the directory-like database that parents tables in other servers. The

special string ':memory:' creates the database in memory only; this is fast and useful for testing but will lose data when your program terminates or if your computer goes down.

For the next example, let's make a database called `enterprise.db` and the table `zoo` to manage our thriving roadside petting zoo business. The table columns are as follows:

`critter`

A variable length string, and our primary key

`count`

An integer count of our current inventory for this animal

NoSQL Data Stores

Some databases are not relational and don't support SQL. These were written to handle very large data sets, allow more flexible data definitions, or support custom data operations. They've been collectively labeled NoSQL (formerly meaning no SQL; now the less confrontational not only SQL).

The dbm Family

The dbm formats were around long before NoSQL was coined. They're key-value stores, often embedded in applications such as web browsers to maintain various settings. A dbm database is like a Python dictionary in the following ways:

- You can assign a value to a key, and it's automatically saved to the database on disk.
- You can get a value from a key.

The following is a quick example. The second argument to the following `open()` method is 'r' to read, 'w' to write, and 'c' for both, creating the file if it doesn't exist:

204 | Chapter 8: Data Has to Go Somewhere

```
>>> import dbm
```

```
>>> db = dbm.open('definitions', 'c')
```

To create key-value pairs, just assign a value to a key just as you would a dictionary:

```
>>> db['mustard'] = 'yellow'
```

```
>>> db['ketchup'] = 'red'
```

```
>>> db['pesto'] = 'green'
```

Let's pause and check what we have so far:

```
>>> len(db)
```

3

```
>>> db['pesto']
```

```
b'green'
```

Now close, then reopen to see if it actually saved what we gave it:

```
>>> db.close()
```

```
>>> db = dbm.open('definitions', 'r')
```

```
>>> db['mustard']
```

```
b'yellow'
```

Keys and values are stored as bytes. You cannot iterate over the database object `db`, but you can get the number of keys by using `len()`. Note that `get()` and `setdefault()` work as they do for dictionaries.

UNIT IV

Python is a particularly good language for web work at every level:

- Clients, to access remote sites
- Servers, to provide data for websites and web APIs
- Web APIs and services, to interchange data in other ways than viewable web pages

Web Clients

The low-level network plumbing of the Internet is called Transmission Control Protocol/Internet Protocol, or more commonly, simply TCP/IP. It moves bytes among computers, but doesn't care about what those bytes mean. That's the job of higher-level protocols—syntax definitions for specific purposes. HTTP is the standard protocol for web data interchange.

The Web is a client-server system. The client makes a request to a server: it opens a TCP/IP connection, sends the URL and other information via HTTP, and receives a response. The format of the response is also defined by HTTP. It includes the status of the request, and (if the request succeeded) the response's data and format.

Test with telnet

HTTP is a text-based protocol, so you can actually type it yourself for web testing. The ancient telnet program lets you connect to any server and port and type commands.

Let's ask everyone's favorite test site, Google, some basic information about its home page. Type this:

```
$ telnet www.google.com 80
```

If there is a web server on port 80 at google.com (I think that's a safe bet), telnet will print some reassuring information and then display a final blank line that's your cue to type something else:

```
Trying 74.125.225.177...
```

```
Connected to www.google.com.
```

```
Escape character is '^]'
```

Now, type an actual HTTP command for telnet to send to the Google web server. The most common HTTP command (the one your browser uses when you type a URL in its location bar)

is GET. This retrieves the contents of the specified resource, such as an HTML file, and returns it to the client. For our first test, we'll use the HTTP command

HEAD, which just retrieves some basic information about the resource:

```
HEAD / HTTP/1.1
```

That HEAD / sends the HTTP HEAD verb (command) to get information about the home page (/).

Python's Standard Web Libraries

In Python 2, web client and server modules were a bit scattered. One of the Python 3 goals was to bundle these modules into two packages

- http manages all the client-server HTTP details:
 - client does the client-side stuff
 - server helps you write Python web servers
 - cookies and cookiejar manage cookies, which save data between site visits
- urllib runs on top of http:
 - request handles the client request
 - response handles the server response
 - parse cracks the parts of a URL

Web Servers

Web developers have found Python to be an excellent language for writing web servers and server-side programs. This has led to such a variety of Python-based web frameworks that it can be hard to navigate among them and make choices

A web framework provides features with which you can build websites, so it does more than a simple web (HTTP) server. You'll see features such as routing (URL to server function), templates (HTML with dynamic inclusions), debugging, and more. I'm not going to cover all of the frameworks here—just those that I've found to be relatively simple to use and suitable for real websites. I'll also show how to run the dynamic parts of a website with Python and other parts with a traditional web server.

The Simplest Python Web Server

You can run a simple web server by typing just one line of Python:


```
$ python -m http.server
```

This implements a bare-bones Python HTTP server. If there are no problems, this will print an initial status message:

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

That 0.0.0.0 means any TCP address, so web clients can access it no matter what address the server has.

```
127.0.0.1 - - [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

localhost and 127.0.0.1 are TCP synonyms for your local computer, so this works regardless of whether you're connected to the Internet. You can interpret this line as follows:

- 127.0.0.1 is the client's IP address
- The first "-" is the remote username, if found

Web Servers | 223

- The second "-" is the login username, if required
- [20/Feb/2013 22:02:37] is the access date and time
- "GET / HTTP/1.1" is the command sent to the web server:
 - The HTTP method (GET)
 - The resource requested (/, the top)
 - The HTTP version (HTTP/1.1)
- The final 200 is the HTTP status code returned by the web server

Web Server Gateway Interface

All too soon, the allure of serving simple files wears off, and we want a web server that can also run programs dynamically. In the early days of the Web, the Common Gateway Interface (CGI) was designed for clients to make web servers run external programs and return the results. CGI also handled getting input arguments from the client through the server to the external programs. However, the programs were started anew for each client access. This could not scale well, because even small programs have appreciable startup time.

Web Services and Automation

We've just looked at traditional web client and server applications, consuming and generating HTML pages. Yet the Web has turned out to be a powerful way to glue applications and data in many more formats than HTML.

The webbrowser Module

Let's start begin a little surprise. Start a Python session in a terminal window and type the following:

```
>>> import antigravity
```

This secretly calls the standard library's webbrowser module and directs your browser to an enlightening Python link.¹

You can use this module directly. This program loads the main Python site's page in your browser:

```
>>> import webbrowser
```

```
>>> url = 'http://www.python.org/'
```

```
>>> webbrowser.open(url)
```

True

This opens it in a new window:

```
>>> webbrowser.open_new(url)
```

True

And this opens it in a new tab, if your browser supports tabs:

```
>>> webbrowser.open_new_tab('http://www.python.org/')
```

True

The webbrowser makes your browser do all the work. Web APIs and Representational State Transfer Often, data is only available within web pages. If you want to access it, you need to access the pages through a web browser and read it. If the authors of the website made any changes since the last time you visited, the location and style of the data might have changed.

Instead of publishing web pages, you can provide data through a web application programming interface (API). Clients access your service by making requests to URLs and getting back responses containing status and data. Instead of HTML pages, the data is in formats that are easier for programs to consume, such as JSON or XML

POST

This verb updates data on the server. It's often used by HTML forms and web APIs.

PUT

This verb creates a new resource.

DELETE

This one speaks for itself: DELETE deletes. Truth in advertising!

Files

Python, like many other languages, patterned its file operations after Unix. Some functions, such as `chown()` and `chmod()`, have the same names, but there are a few new ones.

Create with `open()`

“File Input/Output” on page 173 introduced you to the `open()` function and explains how you can use it to open a file or create one if it doesn't already exist. Let's create a text file called `oops.txt`:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

Check Existence with `exists()`

To verify whether the file or directory is really there or you just imagined it, you can provide `exists()`, with a relative or absolute pathname, as demonstrated here:

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

Check Type with isfile()

The functions in this section check whether a name refers to a file, directory, or symbolic link .

The first function we'll look at, isfile, asks a simple question: is it a plain old lawabiding file?

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
```

True

Here's how you determine a directory:

```
>>> os.path.isdir(name)
```

False

A single dot (.) is shorthand for the current directory, and two dots (..) stands for the parent directory. These always exist, so a statement such as the following will always report True:

```
>>> os.path.isdir('.')
```

True

The os module contains many functions dealing with pathnames (fully qualified filenames, starting with / and including all parents). One such function, isabs(), determines whether its argument is an absolute pathname. The argument doesn't need to be the name of a real file:

```
>>> os.path.isabs(name)
```

False

```
>>> os.path.isabs('/big/fake/name')
```

True

```
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
```

False

Copy with copy()

The copy() function comes from another module, shutil. This example copies the file oops.txt to the file ohno.txt:

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

The `shutil.move()` function copies a file and then removes the original.

Change Name with `rename()`

This function does exactly what it says. In the example here, it renames `ohno.txt` to `ohwell.txt`:

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

Link with `link()` or `symlink()`

In Unix, a file exists in one place, but it can have multiple names, called links. In lowlevel hard links, it's not easy to find all the names for a given file. A symbolic link is an alternative method that stores the new name as its own file, making it possible for you to get both the original and new names at once. The `link()` call creates a hard link, and `symlink()` makes a symbolic link. The `islink()` function checks whether the file is a symbolic link.

Here's how to make a hard link to the existing file `oops.txt` from the new file `yikes.txt`:

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
```

True

To create a symbolic link to the existing file `oops.txt` from the new file `jeepers.txt`, use the following:

```
>>> os.path.islink('yikes.txt')
False
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
```

True

Change Permissions with `chmod()`

On a Unix system, `chmod()` changes file permissions. There are read, write, and execute permissions for the user (that's usually you, if you created the file), the main group that the user is in, and the rest of the world. The command takes an intensely compressed octal (base 8) value that combines user, group, and other permissions. For instance, to make `oops.txt` only readable by its owner, type the following:

```
>>> os.chmod('oops.txt', 0o400)
```

If you don't want to deal with cryptic octal values and would rather deal with (slightly) obscure cryptic symbols, you can import some constants from the stat module and use a statement such as the following:

```
>>> import stat
```

```
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

Change Ownership with `chown()`

This function is also Unix/Linux/Mac-specific. You can change the owner and/or group ownership of a file by specifying the numeric user ID (uid) and group ID (gid):

```
>>> uid = 5
```

```
>>> gid = 22
```

```
>>> os.chown('oops', uid, gid)
```

Get a Pathname with `abspath()`

This function expands a relative name to an absolute one. If your current directory is `/usr/gaberlunzie` and the file `oops.txt` is there, also, you can type the following:

```
>>> os.path.abspath('oops.txt')
```

```
'/usr/gaberlunzie/oops.txt'
```

Get a symlink Pathname with `realpath()`

In one of the earlier sections, we made a symbolic link to `oops.txt` from the new file `jeepers.txt`. In circumstances such as this, you can get the name of `oops.txt` from `jeepers.txt` by using the `realpath()` function, as shown here:

```
>>> os.path.realpath('jeepers.txt')
```

```
'/usr/gaberlunzie/oops.txt'
```

Delete a File with `remove()`

In this snippet, we use the `remove()` function and say farewell to `oops.txt`:

```
>>> os.remove('oops.txt')
```

```
>>> os.path.exists('oops.txt')
```

```
False
```

Directories

In most operating systems, files exist in a hierarchy of directories (more often called folders these days). The container of all of these files and directories is a file system

(sometimes called a volume). The standard `os` module deals with operating specifics such as these and provides the following functions with which you can manipulate them.

Create with `mkdir()`

This example shows how to create a directory called `poems` to store that precious verse:

```
>>> os.mkdir('poems')
```

```
>>> os.path.exists('poems')
```

```
True
```

Delete with `rmdir()`

Upon second thought, you decide you don't need that directory after all. Here's how to delete it:

```
>>> os.rmdir('poems')
```

```
>>> os.path.exists('poems')
```

```
False
```

List Contents with `listdir()`

Okay, take two; let's make `poems` again, with some contents:

```
>>> os.mkdir('poems')
```

Now, get a list of its contents (none so far):

```
>>> os.listdir('poems')
```

```
[]
```

Next, make a subdirectory:

```
>>> os.mkdir('poems/mcintyre')
```

```
>>> os.listdir('poems')
```

```
['mcintyre']
```

Create a file in this subdirectory (don't type all these lines unless you really feel poetic; just make sure you begin and end with matching quotes, either single or tripled):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
```

```
>>> fout.write("Cheerful and happy was his mood,
```

```
... He to the poor was kind and good,
```

```
... And he oft' times did find them food,
```

```
... Also supplies of coal and wood,
```

... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ")

344

```
>>> fout.close()
```

Finally, let's see what we have. It had better be there:

```
>>> os.listdir('poems/mcintyre')
```

```
['the_good_man']
```

Change Current Directory with `chdir()`

With this function, you can go from one directory to another. Let's leave the current directory and spend a little time in poems:

```
>>> import os
```

```
>>> os.chdir('poems')
```

```
>>> os.listdir('.')
```

```
['mcintyre']
```

List Matching Files with `glob()`

The `glob()` function matches file or directory names by using Unix shell rules rather than the more complete regular expression syntax. Here are those rules:

- `*` matches everything (re would expect `.*`)
- `?` matches a single character
- `[abc]` matches character a, b, or c
- `[!abc]` matches any character except a, b, or c

Try getting all files or directories that begin with m:

```
>>> import glob
```

```
>>> glob.glob('m*')
```

```
['mcintyre']
```


How about any two-letter files or directories?

```
>>> glob.glob('??')
```

```
[]
```

I'm thinking of an eight-letter word that begins with m and ends with e:

```
>>> glob.glob('m?????e')
```

```
['mcintyre']
```

What about anything that begins with a k, l, or m, and ends with e?

```
>>> glob.glob('[klm]*e')
```

```
['mcintyre']
```

Programs and Processes

When you run an individual program, your operating system creates a single process. It uses system resources (CPU, memory, disk space) and data structures in the operating system's kernel (file and network connections, usage statistics, and so on). A process is isolated from other processes—it can't see what other processes are doing or interfere with them.

The operating system keeps track of all the running processes, giving each a little time to run and then switching to another, with the twin goals of spreading the work around fairly and being responsive to the user. You can see the state of your processes with graphical interfaces such as the Mac's Activity Monitor (OS X), or Task Manager on Windows-based computers.

You can also access process data from your own programs. The standard library's `os` module provides a common way of accessing some system information. For instance, the following functions get the process ID and the current working directory of the running Python interpreter:

```
>>> import os
```

```
>>> os.getpid()
```

```
76051
```

```
>>> os.getcwd()
```

```
'/Users/williamlubanovic'
```

And these get my user ID and group ID:

```
>>> os.getuid()
```

```
501
```

```
>>> os.getgid()
```

```
20
```

Create a Process with subprocess

All of the programs that you've seen here so far have been individual processes. You can start and stop other existing programs from Python by using the standard library's `subprocess` module. If you just want to run another program in a shell and grab whatever output it created (both standard output and standard error output), use the `getoutput()` function. Here, we'll get the output of the Unix `date` program:

```
>>> import subprocess
```

```
>>> ret = subprocess.getoutput('date')
```

```
>>> ret
```

```
'Sun Mar 30 22:54:37 CDT 2014'
```

You won't get anything back until the process ends. If you need to call something that might take a lot of time, Because the argument to `getoutput()` is a string representing a complete shell command, you can include arguments, pipes, `<` and `>` I/O redirection, and so on:

```
>>> ret = subprocess.getoutput('date -u')
```

```
>>> ret
```

```
'Mon Mar 31 03:55:01 UTC 2014'
```

Piping that output string to the `wc` command counts one line, six “words,” and 29 characters:

```
>>> ret = subprocess.getoutput('date -u | wc')
```

```
>>> ret
```

```
' 1 6 29'
```

A variant method called `check_output()` takes a list of the command and arguments. By default it only returns standard output as type bytes rather than a string and does not use the shell:

```
>>> ret = subprocess.check_output(['date', '-u'])
```

```
>>> ret
```

```
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

To show the exit status of the other program, `getstatusoutput()` returns a tuple with the status code and output:

```
>>> ret = subprocess.getstatusoutput('date')
```

```
>>> ret
```

```
(0, 'Sat Jan 18 21:36:23 CST 2014')
```

If you don't want to capture the output but might want to know its exit status, use `call()`:

```
>>> ret = subprocess.call('date')
```

```
Sat Jan 18 21:33:11 CST 2014
```

```
>>> ret
```

```
0
```

(In Unix-like systems, 0 is usually the exit status for success.)

That date and time was printed to output but not captured within our program. So, we saved the return code as `ret`.

You can run programs with arguments in two ways. The first is to specify them in a single string. Our sample command is `date -u`, which prints the current date and time in UTC (you'll read more about UTC in a few pages):

```
>>> ret = subprocess.call('date -u', shell=True)
```

```
Tue Jan 21 04:40:04 UTC 2014
```

You need that `shell=True` to recognize the command line `date -u`, splitting it into separate strings and possibly expanding any wildcard characters such as `*` (we didn't use any in this example).

The second method makes a list of the arguments, so it doesn't need to call the shell:

```
>>> ret = subprocess.call(['date', '-u'])
```

```
Tue Jan 21 04:41:59 UTC 2014
```

Create a Process with multiprocessing

You can run a Python function as a separate process or even run multiple independent processes in a single program with the multiprocessing module. Here's a short example that does nothing useful; save it as `mp.py` and then run it by typing `python mp.py`:

```
import multiprocessing
```

```
import os
```

```

def do_this(what):
    whoami(what)
def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))
if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
            args=("I'm function %s" % n,))
        p.start()

```

When I run this, my output looks like this:

```
Process 6224 says: I'm the main program
```

```
Process 6225 says: I'm function 0
```

```
Process 6226 says: I'm function 1
```

```
Process 6227 says: I'm function 2
```

```
Process 6228 says: I'm function 3
```

The Process() function spawned a new process and ran the do_this() function in it. Because we did this in a loop that had four passes, we generated four new processes that executed do_this() and then exited. The multiprocessing module has more bells and whistles than a clown on a calliope. It's really intended for those times when you need to farm out some task to multiple processes to save overall time; for example, downloading web pages for scraping, resizing images, and so on. It includes ways to queue tasks, enable intercommunication among processes,

Kill a Process with terminate()

If you created one or more processes and want to terminate one for some reason (perhaps it's stuck in a loop, or maybe you're bored, or you want to be an evil overlord), use terminate(). In the example that follows, our process would count to a million, sleeping at each step for a second, and printing an irritating message. However, our main program runs out of patience in five seconds and nukes it from orbit:

```
import multiprocessing
```

```

import time
import os
def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))
def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)
if __name__ == "__main__":
    whoami("main")
    p = multiprocessing.Process(target=loopy, args=("loopy",))
    p.start()
    time.sleep(5)
    p.terminate()

```

When I run this program, I get the following:

I'm main, in process 97080

I'm loopy, in process 97081

Number 1 of 1000000. Honk!

Number 2 of 1000000. Honk!

Number 3 of 1000000. Honk!

Number 4 of 1000000. Honk!

Number 5 of 1000000. Honk!

Calendars and Clocks

Programmers devote a surprising amount of effort to dates and times. Let's talk about some of the problems they encounter, and then get to some best practices and tricks to make the situation a little less messy.

Dates can be represented in many ways—too many ways, actually. Even in English with

the Roman calendar, you'll see many variants of a simple date:

- July 29 1984
- 29 Jul 1984
- 29/7/1984
- 7/29/1984

Among other problems, date representations can be ambiguous. In the previous examples, it's easy to determine that 7 stands for the month and 29 is the day of the month,

largely because months don't go to 29. But how about 1/6/2012? Is that referring to January 6 or June 1? The month name varies by language within the Roman calendar. Even the year and

month can have a different definition in other cultures Leap years are another wrinkle. You probably know that every four years is a leap year (and the summer Olympics and the American presidential election). Did you also know that every 100 years is not a leap year, but that every 400 years is? Here's code to test

various years for leapiness:

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

Times have their own sources of grief, especially because of time zones and daylight savings time. If you look at a time zone map, the zones follow political and historic boundaries rather than every 15 degrees (360 degrees / 24) of longitude. And countries start and end daylight

saving times on different days of the year. In fact, countries in the southern hemisphere advance their clocks when the northern hemisphere is winding them back, and vice versa.

Python's standard library has many date and time modules: `datetime`, `time`, `calendar`, `dateutil`, and others. There's some overlap, and it's a bit confusing.

The `datetime` Module

Let's begin by investigating the standard `datetime` module. It defines four main objects, each with many methods:

- `date` for years, months, and days
- `time` for hours, minutes, seconds, and fractions
- `datetime` for dates and times together
- `timedelta` for date and/or time intervals

You can make a date object by specifying a year, month, and day. Those values are then available as attributes:

```
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> halloween
datetime.date(2014, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2014
```

You can print a date with its `isoformat()` method:

```
>>> halloween.isoformat()
'2014-10-31'
```

The iso refers to ISO 8601, an international standard for representing dates and times.

It goes from most general (year) to most specific (day). It also sorts correctly: by year, then month, then day. I usually pick this format for date representation in programs, and for filenames that save data by date. The next section describes the more complex

strptime() and strftime() methods for parsing and formatting dates.

This example uses the today() method to generate today's date:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2014, 2, 2)
```

This one makes use of a timedelta object to add some time interval to a date:

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
datetime.date(2014, 2, 3)
>>> now + 17*one_day
datetime.date(2014, 2, 19)
>>> yesterday = now - one_day
>>> yesterday
datetime.date(2014, 2, 1)
```

The range of date is from date.min (year=1, month=1, day=1) to date.max (year=9999, month=12, day=31). As a result, you can't use it for historic or astronomical calculations.

The datetime module's time object is used to represent a time of day:

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
252 | Chapter 10: Systems
>>> noon.minute
0
>>> noon.second
```



```
0
```

```
>>> noon.microsecond
```

```
0
```

The arguments go from the largest time unit (hours) to the smallest (microseconds). If you don't provide all the arguments, time assumes all the rest are zero. By the way, just because you can store and retrieve microseconds doesn't mean you can retrieve time from your computer to the exact microsecond. The accuracy of subsecond measurements depends on many factors in the hardware and operating system.

The datetime object includes both the date and time of day. You can create one directly, such as the one that follows, which is for January 2, 2014, at 3:04 A.M., plus 5 seconds and 6 microseconds:

```
>>> from datetime import datetime
```

```
>>> some_day = datetime(2014, 1, 2, 3, 4, 5, 6)
```

```
>>> some_day
```

```
datetime.datetime(2014, 1, 2, 3, 4, 5, 6)
```

The datetime object also has an isoformat() method:

```
>>> some_day.isoformat()
```

```
'2014-01-02T03:04:05.000006'
```

That middle T separates the date and time parts.

datetime has a now() method with which you can get the current date and time:

```
>>> from datetime import datetime
```

```
>>> now = datetime.now()
```

```
>>> now
```

```
datetime.datetime(2014, 2, 2, 23, 15, 34, 694988)
```

```
14
```

```
>>> now.month
```

```
2
```

```
>>> now.day
```

```
2
```

```
>>> now.hour
```

23

```
>>> now.minute
```

15

```
>>> now.second
```

34

```
>>> now.microsecond
```

694988

You can merge a date object and a time object into a datetime object by using `combine()`:

1. This starting point is roughly when Unix was born.

```
>>> from datetime import datetime, time, date
```

```
>>> noon = time(12)
```

```
>>> this_day = date.today()
```

```
>>> noon_today = datetime.combine(this_day, noon)
```

```
>>> noon_today
```

```
datetime.datetime(2014, 2, 2, 12, 0)
```

You can yank the date and time from a datetime by using the `date()` and `time()` methods:

```
>>> noon_today.date()
```

```
datetime.date(2014, 2, 2)
```

```
>>> noon_today.time()
```

```
datetime.time(12, 0)
```

Using the time Module

It is confusing that Python has a datetime module with a time object, and a separate time module. Furthermore, the time module has a function called—wait for it—`time()`.

One way to represent an absolute time is to count the number of seconds since some starting point. Unix time uses the number of seconds since midnight on January 1, 1970.1

This value is often called the epoch, and it is often the simplest way to exchange dates and times among systems.

The time module's `time()` function returns the current time as an epoch value:

```
>>> import time
>>> now = time.time()
>>> now
1391488263.664645
```

If you do the math, you'll see that it has been over one billion seconds since New Year's, 1970. Where did the time go?

You can convert an epoch value to a string by using `ctime()`:

```
>>> time.ctime(now)
'Mon Feb 3 22:31:03 2014'
```

In the next section, you'll see how to produce more attractive formats for dates and times.

Epoch values are a useful least-common denominator for date and time exchange with different systems, such as JavaScript. Sometimes, though, you need actual days, hours, and so forth, which time provides as `struct_time` objects. `localtime()` provides the time in your system's time zone, and `gmtime()` provides it in UTC:

```
>>> time.localtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=3, tm_hour=22, tm_min=31,
tm_sec=3, tm_wday=0, tm_yday=34, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=4, tm_min=31,
tm_sec=3, tm_wday=1, tm_yday=35, tm_isdst=0)
```

In my (Central) time zone, 22:31 was 04:31 of the next day in UTC (formerly called Greenwich time or Zulu time). If you omit the argument to `localtime()` or `gmtime()`, they assume the current time.

The opposite of these is `mktime()`, which converts a `struct_time` object to epoch seconds:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1391488263.0
```

This doesn't exactly match our earlier epoch value of `now()` because the `struct_time` object preserves time only to the second.

Some advice: wherever possible, use UTC instead of time zones. UTC is an absolute time, independent of time zones. If you have a server, set its time to UTC; do not use local time.

Here's some more advice (free of charge, no less): never use daylight savings time if you can avoid it. If you use daylight savings time, an hour disappears at one time of year ("spring ahead") and occurs twice at another time ("fall back"). For some reason, many organizations use daylight savings in their computer systems, but are mystified every year by data duplicates and dropouts. It all ends in tears.

Remember, your friends are UTC for times, and UTF-8 for strings

Read and Write Dates and Times

`isoformat()` is not the only way to write dates and times. You already saw the `ctime()` function in the `time` module, which you can use to convert epochs to strings:

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
```

```
'Mon Feb 3 21:14:36 2014'
```

You can also convert dates and times to strings by using `strftime()`. This is provided as a method in the `datetime`, `date`, and `time` objects, and as a function in the `time` module. `strftime()` uses format strings to specify the output,

UNIT V

Concurrency

The official Python site discusses concurrency in general and in the standard library.

Those pages have many links to various packages and techniques; we'll show the most useful ones in this chapter.

In computers, if you're waiting for something, it's usually for one of two reasons:

I/O bound

This is by far more common. Computer CPUs are ridiculously fast—hundreds of times faster than computer memory and many thousands of times faster than disks or networks.

CPU bound

This happens with number crunching tasks such as scientific or graphic calculations.

Two more terms are related to concurrency:

synchronous

One thing follows the other, like a funeral procession.

asynchronous

Tasks are independent, like party-goers dropping in and tearing off in separate cars.

As you progress from simple systems and tasks to real-life problems, you'll need at some point to deal with concurrency. Consider a website, for example. You can usually provide static and dynamic pages to web clients fairly quickly. A fraction of a second is considered interactive, but if the display or interaction takes longer, people become impatient.

Tests by companies such as Google and Amazon showed that traffic drops off quickly if the page loads even a little slower.

Queues

A queue is like a list: things are added at one end and taken away from the other. The most common is referred to as FIFO (first in, first out).

Suppose that you're washing dishes. If you're stuck with the entire job, you need to wash each dish, dry it, and put it away. You can do this in a number of ways. You might wash the first dish, dry it, and then put it away. You then repeat with the second dish, and so on. Or, you might batch operations and wash all the dishes, dry them all, and then put them away; this assumes you have space in your sink and drainer for all the dishes that

accumulate at each step. These are all synchronous approaches—one worker, one thing at a time.

Processes

You can implement queues in many ways. For a single machine, the standard library's multiprocessing module (which you can see in “Programs and Processes” on page 247) contains a Queue function. Let's simulate just a single washer and multiple dryer processes (someone can put the dishes away later) and an intermediate dish_queue.

Call this program dishes.py:

```
import multiprocessing as mp
def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)
def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
        input.task_done()
dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()
dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Run your new program thusly:

```
$ python dishes.py
```

```
Washing salad dish
```

```
Washing bread dish
```

```
Washing entree dish
```

```
Washing dessert dish
```

Drying salad dish

Drying bread dish

Drying entree dish

Drying dessert dish

Threads

A thread runs within a process with access to everything in the process, similar to a multiple personality. The multiprocessing module has a cousin called threading that uses threads instead of processes (actually, multiprocessing was designed later as its process-based counterpart). Let's redo our process example with threads:

```
import threading
def do_this(what):
    whoami(what)
def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))
if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
            args=("I'm function %s" % n,))
        p.start()
```

Here's what prints for me:

```
Thread <_MainThread(MainThread, started
```

Network

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This chapter gives you understanding on most famous concept in Networking - Socket Programming.

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary –

Sr.No.	Term & Description
1	Domain The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
2	type The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
3	protocol Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	hostname The identifier of a network interface – <ul style="list-style-type: none">• A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation• A string "<broadcast>", which specifies an INADDR_BROADCAST address.• A zero-length string, which specifies INADDR_ANY, or• An Integer, interpreted as a binary address in host byte order.

5	<p>port</p> <p>Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.</p>
---	--

The *socket* Module

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax –

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Here is the description of the parameters –

- **socket_family** – This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type** – This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol** – This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required –

Server Socket Methods

Sr.No.	Method & Description
1	<p>s.bind()</p> <p>This method binds address (hostname, port number pair) to socket.</p>
2	<p>s.listen()</p> <p>This method sets up and start TCP listener.</p>
3	<p>s.accept()</p> <p>This passively accept TCP client connection, waiting until connection arrives (blocking).</p>

Client Socket Methods

Sr.No.	Method & Description
1	<p>s.connect()</p> <p>This method actively initiates TCP server connection.</p>

General Socket Methods

Sr.No.	Method & Description
1	s.recv() This method receives TCP message
2	s.send() This method transmits TCP message
3	s.recvfrom() This method receives UDP message
4	s.sendto() This method transmits UDP message
5	s.close() This method closes socket
6	socket.gethostname() Returns the hostname.

A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python      # This is server.py file

import socket         # Import socket module

s = socket.socket()   # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345         # Reserve a port for your service.
s.bind((host, port)) # Bind to the port
```

```
s.listen(5)          # Now wait for client connection.
while True:
    c, addr = s.accept() # Establish connection with client.
    print 'Got connection from', addr
    c.send('Thank you for connecting')
    c.close()         # Close the connection
```

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The **socket.connect(hostname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits –

```
#!/usr/bin/python    # This is client.py file

import socket        # Import socket module

s = socket.socket()  # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345         # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close()           # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

Following would start a server in background.

```
$ python server.py &
```

Once server is started run client as follows:

```
$ python client.py
```

This would produce following result –

Got connection from ('127.0.0.1', 48437)

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Thank you for connecting

Python Internet modules

A list of some important modules in Python Network/Internet programming.

Internet Services

Python has an extensive networking toolset. In the following sections, we'll look at ways to automate some of the most popular Internet services. The official, comprehensive documentation is available online.

Domain Name System

Computers have numeric IP addresses such as 85.2.101.94, but we remember names better than numbers. The Domain Name System (DNS) is a critical Internet service that converts IP addresses to and from names via a distributed database. Whenever you're using a web browser and suddenly see a message like "looking up host," you've probably lost your Internet connection, and your first clue is a DNS failure.

Some DNS functions are found in the low-level socket module. `gethostbyname()`

returns the IP address for a domain name, and the extended edition `gethostbyname_ex()` returns the name, a list of alternative names, and a list of addresses:

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

The `getaddrinfo()` method looks up the IP address, but it also returns enough information to create a socket to connect to it:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
[(2, 2, 17, "", ('66.6.44.4', 80)), (2, 1, 6, "", ('66.6.44.4', 80))]
```

The preceding call returned two tuples, the first for UDP, and the second for TCP (the 6 in the 2, 1, 6 is the value for TCP).

You can ask for TCP or UDP information only:

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,
socket.SOCK_STREAM)
[(2, 1, 6, "", ('66.6.44.4', 80))]
```

Some TCP and UDP port numbers are reserved for certain services by IANA, and are associated with service names. For example, HTTP is named `http` and is assigned TCP port 80.

These functions convert between service names and port numbers:

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```

290 | Chapter 11: Concurrency and Networks

Python Email Modules

The standard library contains these email modules:

- `smtplib` for sending email messages via Simple Mail Transfer Protocol (SMTP)

- email for creating and parsing email messages
- poplib for reading email via Post Office Protocol 3 (POP3)
- imaplib for reading email via Internet Message Access Protocol (IMAP)

The official documentation contains sample code for all of these libraries.

If you want to write your own Python SMTP server, try smtpd.

A pure-python SMTP server called Lamson allows you to store messages in databases, and you can even block spam.

Other protocols

Using the standard ftplib module, you can push bytes around by using the File Transfer Protocol (FTP). Although it's an old protocol, FTP still performs very well.

You've seen many of these modules in various places in this book, but also try the documentation for standard library support of Internet protocols.

REFERENCES

1. TUTORIAL POINT <https://www.tutorialspoint.com/python/index.htm>
Bill Lubanovic, "Introducing Python", O'Reilly, First Edition-Second Release, 2014.
- 2.