Class: 5th Semester Computer Engg.

SUBJECT: COMPUTER PROGRAMMING USING PYTHON

CHAPTER – 1 (INTRODUCTION)

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. 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 C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Advantages of Python: Some of the key advantages of learning Python are:

- **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.

Characteristics of Python:

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, ActiveX, CORBA, and Java.

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, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

OVERVIEW

Getting Python: The most up-to-date and current source code, documentation, news, etc., is available on the official website of Python https://www.python.org/. You can download Python documentation from https://www.python.org/doc/**.**

Installing Python: Python distribution is available for a wide variety of platforms.

Windows Installation: Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Setting up PATH: Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

- To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

C:\Users\Your Name> python --version

Running Python: There are three different ways to start Python −

1. Interactive Interpreter: You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** at the command line.

C:> python

Start coding right away in the interactive interpreter.

>>> print(“Hello, World!”)

Which will write "Hello, World!" in the command line:

2. Script from the Command-line: Python is an interpreted programming language, this means that as a developer you write Python (.py) files in a text editor and then put those files into the python interpreter to be executed. The way to run a python file is like this on the command line:

C:\Users\Your Name> python helloworld.py

Where "helloworld.py" is the name of your python file.

Let's write our first Python file, called helloworld.py, which can be done in any text editor.

```
print("Hello, World!")
```

Save your file. Open your command line, navigate to the directory where you saved your file, and run:

C:\Users\Your Name>python helloworld.py

The output should read:

Hello, World!

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

exit()

3. Integrated Development Environment: You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python. Some common IDEs are:

PyCharm, IDLE, Atom, Spyder

First Python Program: Let us execute programs in different modes of programming.

1. Interactive Mode Programming

Simply type the following command by invoking the interpreter to bring the command prompt −

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Type the following text at the Python prompt and press the Enter −

```
>>> print "Hello, Python!"
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**

Hello, Python!

2. Script Mode Programming

Let us write a simple Python program using text editor in a script. Python files have extension **.py**. Type the following source code in a test.py file (simple text file)−

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows −

```
$ python test.py
```

This produces the following result −

Hello, Python!

Python Identifiers: A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words: The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only. These are:

and, assert, break, class, continue, def, del, else, elif, for, if, import, not, or, pass, print, raise, return, try, while, with

Lines and Indentation: Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code. For example:

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation. For example:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you, but it has to be at least one.

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error. For example:

```
if 5 > 2:
 print("Five is greater than two!")
      print("Five is greater than two!")
```

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

IDLE:

It is Python's **Integrated Development and Learning Environment**. It has following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features

CHAPTER – 2 (BASIC PYTHON SYNTAX)

Quotation in Python: Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python: A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
print "Hello, Python!" # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code.

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as multiline comments:

```
'''
This is a multiline
comment.
'''
```

Multiple Statements on a Single Line: The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon

```
import sys;  x = 'foo';   sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites: A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example −

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Variable Names: Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables: Python variables ***do not need explicit declaration*** to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

```
100
1000.0
John
```

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, _)
- Variable names are ***case-sensitive (age, Age and AGE are three different variables)***

```
#Legal variable names:
myvar = "John";   my_var = "John";   _my_var = "John";  myVar = "John";  MYVAR = "John";
myvar2 = "John"

#Illegal variable names:
2myvar = "John";  my-var = "John";  my var = "John"
```

Assign Value to Multiple Variables: Python allows you to assign values to multiple variables in one line. e.g.

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

And you can assign the *same* value to multiple variables in one line.e.g.

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Output Variables: The Python print statement is often used to output variables. To combine both text and a variable, Python uses the + character:

```
x = "awesome"
print("Python is " + x)
```

You can also use the + character to add a variable to another variable:

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

For numbers, the + character works as a mathematical operator:

```
x = 5
y = 10
print(x + y)
```

If you try to combine a string and a number, Python will give you an error:

```
x = 5
y = "John"
print(x + y)
```

Global Variables: Variables that are created outside of a function (as in all of the examples above) are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

- Create a variable outside of a function, and use it inside the function

```
x = "awesome"
def myfunc():
  print("Python is " + x)
myfunc()
```

(Result – Python is awesome)

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value. e.g.

- Create a variable inside a function, with the same name as the global variable

```
x = "awesome"
def myfunc():
  x = "fantastic"
  print("Python is " + x)
myfunc()
print("Python is " + x)
```

(Result – Python is fantastic
Python is awesome)

The global Keyword: Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function. To create a global variable inside a function, you can use the global keyword.

```
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)
```

(Result – Python is fantastic)

Also, use the global keyword if you want to change a global variable inside a function. To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = "awesome"
def myfunc():
  global x
  x = "fantastic"
myfunc()
print("Python is " + x)
```

(Result – Python is fantastic)

Built-in Data Types: Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

Getting the Data Type: You can get the data type of any object by using the type() function:

Print the data type of the variable x:

```
x = 5
print(type(x))
```

(Result – class 'int')

Setting the Data Type: In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Python Casting: There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types. Casting in python is therefore done using constructor functions:

- int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Integers:

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
```

Floats:

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
```

```
w = float("4.2") # w will be 4.2
```

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

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

a) 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 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
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J

- 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.

b) 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
```

c) 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 values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
list = [ 'abcd', 786, 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result −

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

d) ***Python Tuples:*** 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**. Tuples can be thought of as **read-only** lists. For example −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple              # Prints the complete tuple
print tuple[0]           # Prints first element of the tuple
print tuple[1:3]         # Prints elements of the tuple starting from 2nd till 3rd
print tuple[2:]          # Prints elements of the tuple starting from 3rd element
print tinytuple * 2      # Prints the contents of the tuple twice
print tuple + tinytuple  # Prints concatenated tuples
```

This produce the following result −

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

e) ***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.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john', 'code': 6734, 'dept': 'sales'}

print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

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.

CHAPTER-3 (LANGUAGE COMPONENTS)

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

a) Python Arithmetic Operators: Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

b) Python Comparison Operators: These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators. Assume variable a holds 10 and variable b holds 20 −

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.

<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

c) Python Assignment Operators: Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a

d) Python Bitwise Operators: Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

e) Python Logical Operators: There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

f) Python Membership Operators: Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below:

Operator	Description	Example
In	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

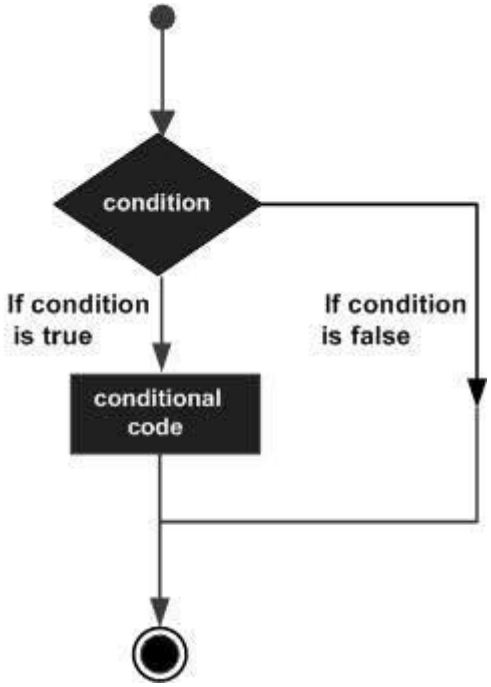
g) Python Identity Operators: Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Decision Making Statements: 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 supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

An "if statement" is written by using the if keyword. Example is:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation: Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example – If statement without indentation (will raise an error)

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif: The elif keyword is python's way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else: The else keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If: If you have only one statement to execute, you can put it on the same line as the if statement.

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else: If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line. This technique is known as **Ternary Operators or Conditional Expression**. Example - One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And: The and keyword is a logical operator, and is used to combine conditional statements:

Example – Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or: The or keyword is a logical operator, and is used to combine conditional statements:

Example – Test is a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If: You can have if statements inside if statements, this is called *nested* if statements.

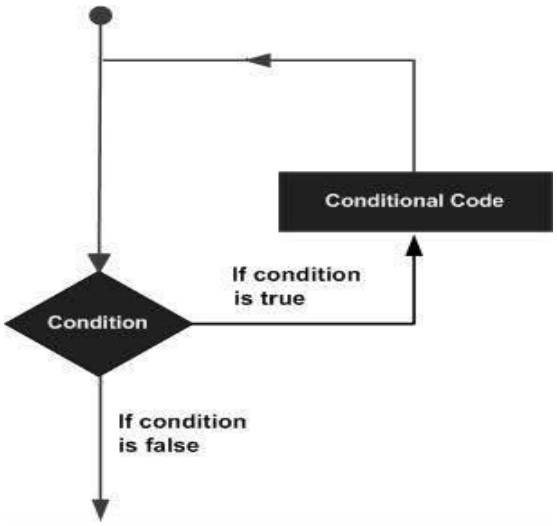
```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

The pass Statement: if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 33
b = 200
if b > a:
  pass
```

WHILE LOOP

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –

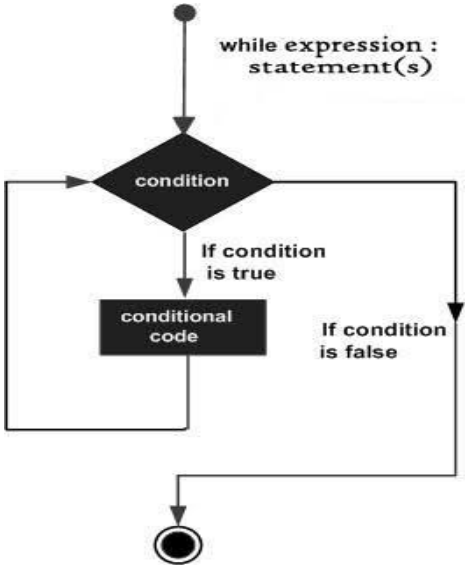


Python programming language provides various loops to handle looping requirements.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true. The syntax of a **while** loop in Python programming language is −

```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.



In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. **Python uses indentation as its method of grouping statements**.

```
count = 0
while (count < 4):
   print 'The count is:', count
   count = count + 1

print "Good bye!"
```

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

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 4. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop: A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
var = 1
while var == 1 :  # This constructs an infinite loop
   num = raw_input("Enter a number  :")
   print "You entered: ", num
print "Good bye!"
```

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

```
Enter a number  :20
You entered:  20
Enter a number  :3
You entered:  3
Enter a number between :Traceback (most recent call last):
   File "test.py", line 5, in <module>
      num = raw_input("Enter a number :")
KeyboardInterrupt
```

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

***Using else Statement with While Loop*:** Python supports to have an **else** statement associated with a loop statement.

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 3, otherwise else statement gets executed.

```
count = 0
while count < 3:
   print count, " is  less than 3"
   count = count + 1
else:
   print count, " is not less than 3"
```

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

```
0 is less than 3
1 is less than 3
```

2 is less than 3
3 is not less than 3

***Single Statement Suites*:** Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header. Here is the syntax and example of a **one-line while** clause −

```
flag = 1
while (flag): print 'Given flag is really true!'
print "Good bye!"
```

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

Nested while loop: The syntax for a **nested while loop** statement in Python programming language is as follows −

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example: Following program uses a nested for loop to find the prime numbers from 2 to 20

```
i = 2
while(i < 20):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print i, " is prime"
   i = i + 1
print "Good bye!"
```

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

2 is prime; 3 is prime; 5 is prime; 7 is prime; 11 is prime; 13 is prime; 17 is prime; 19 is prime
Good bye!

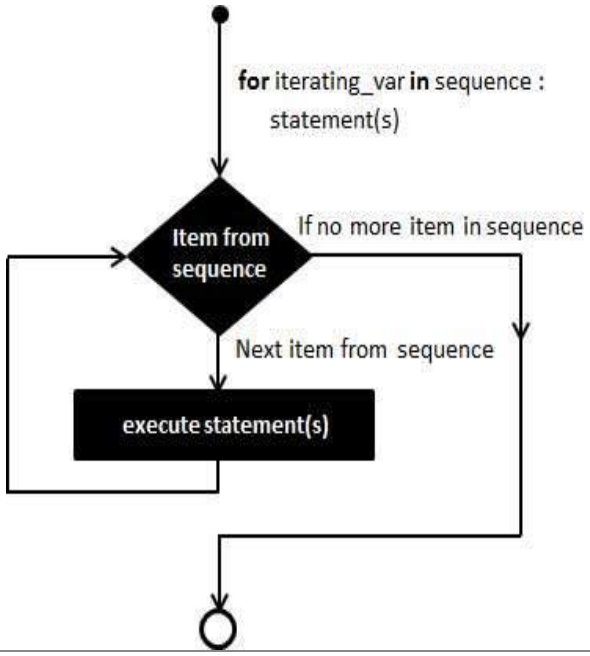
FOR LOOP: It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram:




```
for letter in 'Python':     # First Example
  print 'Current Letter :', letter
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
  print 'Current fruit :', fruit

print "Good bye!"
```

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

Current Letter : P; Current Letter : y; Current Letter : t; Current Letter : h; Current Letter : o
Current Letter : n
Current fruit : banana; Current fruit : apple; Current fruit : mango
Good bye!

Iterating by Sequence Index: An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
  print 'Current fruit :', fruits[index]
print "Good bye!"
```

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

Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with For Loop: Python supports to have an else statement associated with a loop statement

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 17.

```
for num in range(10,17):     #to iterate between 10 to 17
  for i in range(2,num):    #to iterate on the factors of the number
    if num%i == 0:         #to determine the first factor
      j=num/i             #to calculate the second factor
      print '%d equals %d * %d' % (num,i,j)
      break #to move to the next number, the #first FOR
  else:                  # else part of the loop
    print num, 'is a prime number'
                         break
```

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

10 equals 2 * 5; 11 is a prime number; 12 equals 2 * 6; 13 is a prime number
14 equals 2 * 7; 15 equals 3 * 5; 16 equals 2 * 8; 17 is a prime number

The range() Function: To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example: Using the range() function

```
for x in range(6):
  print(x)
```
Output is: 0 1 2 3 4 5

Note that range(6) is not the values of 0 to 6, but the values 0 to 5

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example: Using the start parameter:

```
for x in range(2, 6):
  print(x)
```
Output is: 2 3 4 5

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example: Increment the sequence with 3 (default is 1):

```
for x in range(2, 20, 3):
  print(x)
```
Output is: 2 5 8 11 14 17

Else in For Loop: The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example – Print all numbers from 0 to 3, and print a message when the loop has ended:

```
for x in range(4):
  print(x)
```

```
else:
  print("Finally finished!")
```

Output is: 0 1 2 3 Finally finished!

Nested Loops: A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop":

Example: Print each adjective for every fruit:

```
adj = ["red", "big"]
fruits = ["apple", "banana"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Output is:
red apple; red banana; big apple; big banana

Break Statement: With the break statement we can stop the loop even if the while condition is true: Example – Exit the loop when I is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

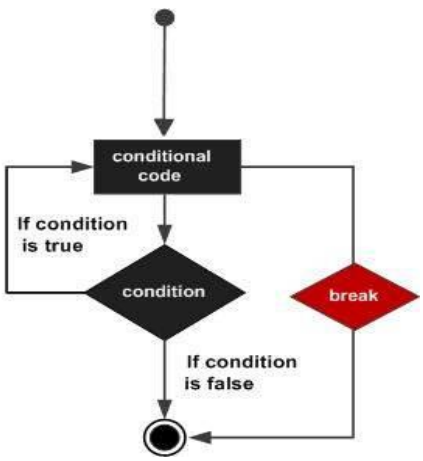
Output is: 1 2 3

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Flow Diagram:



```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print 'Current Letter :', letter
var = 10                    # Second Example
while var > 0:
   print 'Current variable value :', var
   var = var -1
   if var == 5:
      break

print "Good bye!"
```

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

Current Letter : P; Current Letter : y; Current Letter : t
Current variable value : 10; Current variable value : 9; Current variable value : 8
Current variable value : 7; Current variable value : 6
Good bye!

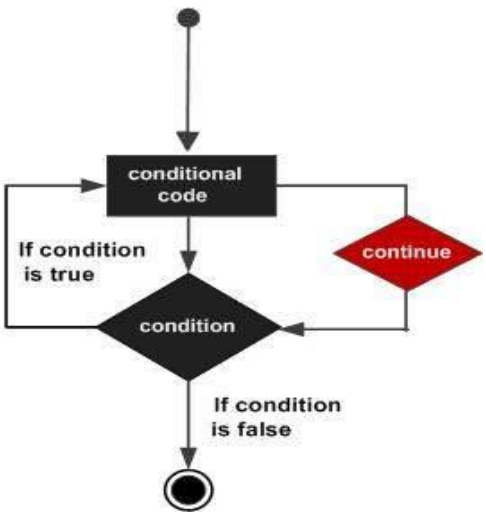
Continue Statement: With the continue statement we can stop the current iteration, and continue with the next. Example – Continue to the next iteration if I is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Output is: # Note that number 3 is missing in the result
1 2 4 5 6

It returns the control to the beginning of the while loop.. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops.

Flow Diagram



```
for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print 'Current Letter :', letter
var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print 'Current variable value :', var
print "Good bye!"
```

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

Current Letter : P; Current Letter : y; Current Letter : t; Current Letter : o; Current Letter : n
Current variable value : 9; Current variable value : 8; Current variable value : 7; Current variable value : 6; Current variable value : 4; Current variable value : 3; Current variable value : 2; Current variable value : 1; Current variable value : 0
Good bye!

Pass Statement: It is used when a statement is required syntactically but you do not want any command or code to execute. The **pass** statement is a *null* operation; nothing happens when it executes. ***Example:***

```
for letter in 'Python':
   if letter == 'h':
      pass
      print 'This is pass block'
   print 'Current Letter :', letter
print "Good bye!"
```

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

Current Letter : P; Current Letter : y; Current Letter : t; This is pass block
Current Letter : h; Current Letter : o; Current Letter : n; Good bye!

CHAPTER-4 (COLLECTIONS)

Python Collections (Arrays): There are 4 collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered and changeable. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

LIST: Lists are used to store multiple items in a single variable. Lists are created using square brackets:

Example: Create a List

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

Result: [‘apple’, ‘banana’, ‘cherry’]

a) List Items: List items are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index [0], the second item has index [1] etc.

b) Ordered: When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

c) Changeable: The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

d) Allow Duplicates: Since lists are indexed, lists can have items with the same value.

Example: Lists allow duplicate values.

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

Result: [‘apple’, ‘banana’, ‘cherry’, ‘apple’, ‘cherry’]

1. List Length: To determine how many items a list has, use the len() function.

Example: Print the number of items in the list.

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Result: 3

2. List Items - Data Types: List items can be of any data type.

Example: String, int and boolean data types

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types.

Example: A list with strings, integers and boolean values.

```
list1 = ["abc", 34, True, 40, "male"]
```

3. type(): From Python's perspective, lists are defined as objects with the data type 'list'.

Example: What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```
Result: class 'list'

4. The list() Constructor: It is also possible to use the list() constructor when creating a new list.

Example: Using the list() constructor to make a list.

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```
Result: ['apple', 'banana', 'cherry']

ACCESSING THE LIST ITEMS: List items are indexed and you can access them by referring to the index number.

Example: Print the second item of the list.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])
```

Result: banana

Note: The first item has index 0.

1. Negative Indexing: Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Example: Print the last item of the list.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Result: cherry

2. Range of Indexes: You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.

Example: Return the third, fourth and fifth item.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
```

```
print(thislist[2:5])
```

Result: [‘cherry’, ‘orange’, ‘kiwi’]

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0. By leaving out the start value, the range will start at the first item.

Example: This returns the items from the beginning to, but NOT included, “kiwi”.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

Result: [‘apple’, ‘banana’, ‘cherry’, ‘orange’]

This will return the items from index 0 to index 4. Remember that index 0 is the first item, and index 4 is the fifth item. Remember that the item in index 4 is NOT included

By leaving out the end value, the range will go on to the end of the list.

Example: This example returns the items from “cherry” and to the end.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

Result: [‘cherry’, ‘orange’, ‘kiwi’, ‘melon’,’mango’]

This will return the items from index 2 to the end. Remember that index 0 is the first item, and index 2 is the third.

3. Range of Negative Indexes: Specify negative indexes if you want to start the search from the end of the list.

Example: This example returns the items from “orange”(-4) to, but NOT included “mango”(-1).

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])
```

Result: [‘orange’, ‘kiwi’, ‘melon’]

Negative indexing means starting from the end of the list. This example returns the items from index -4 (included) to index -1 (excluded). Remember that the last item has the index -1,

4. Check if Item Exists: To determine if a specified item is present in a list use the `in` keyword.

Example: Check if “apple” is present in the list.

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

Result: Yes, ‘apple’ is in the fruits list

CHANGE/ADD/REMOVE LIST ITEMS

1. Change Item Value: To change the value of a specific item, refer to the index number.

Example 1: Change the second item.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```
Result: ['apple', 'blackcurrant', 'cherry']

To insert more than one item, create a list with the new values, and specify the index number where you want the new values to be inserted.

Example 2: Change the second value by replacing it with *two* new values.

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = ["blackcurrant", "watermelon"]
print(thislist)
```
Result: ['apple', ['blackcurrant', 'watermelon'], 'cherry']

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

2. Change a Range of Item Values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example 3: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon".

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Result: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

3. Insert Items: To insert a new list item, without replacing any of the existing values, we can use the insert() method. The insert() method inserts an item at the specified index.

Example 4: Insert "watermelon" as the third item.

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```
Result: ['apple', 'banana', 'watermelon', 'cherry']

Note: As a result of the example above, the list will now contain 4 items.

4. Append Items: To add an item to the end of the list, use the append() method.

Example 5: Using the append() method to append an item.

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Result: [‘apple’, ‘banana’, ‘cherry’, ‘orange’]

5. Remove Specified Item: The remove() method removes the specified item.

Example 6: Remove "banana".

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Result: [‘apple’, ‘cherry’]

6. Remove Specified Index: The pop() method removes the specified index.

Example 7: Remove the second item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

Result: [‘apple’, ‘cherry’]. If you do not specify the index, the pop() method removes the last item.

Example 8: Remove the last item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Result: [‘apple’, ‘banana’]

7. The del keyword also removes the specified index.

Example 9: Remove the first item.

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

Result: [‘banana’, ‘cherry’]

8. The del keyword can also delete the list completely.

Example 10: Delete the entire list.

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Result: Successfully deleted “thislist”

9. Clear the List: The clear() method empties the list. The list still remains, but it has no content.

Example 11: Clear the list content.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

Result: [ ]

LOOP THROUGH A LIST

1. **You can loop through the list items by using for loop.**

Example: Print all items in the list, one by one.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```
Result: apple, banana, cherry

2. **Using a While Loop:** You can loop through the list items by using a while loop. Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by refering to their indexes. Remember to increase the index by 1 after each iteration.

Example: Print all items, using a while loop to go through all the index numbers.

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```
Result: apple, banana, cherry

COPY A LIST: You cannot copy a list simply by typing list2 = list1, because: list2 will only be a *reference* to list1, and changes made in list1 will automatically also be made in list2.

1. **There are ways to make a copy, one way is to use the built-in List method copy().**

Example: Make a copy of a list with the copy() method.

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Result: ['apple', 'banana', 'cherry']

2. **Another way to make a copy is to use the built-in method list().**

Example: Make a copy of a list with the list() method.

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

Result: ['apple', 'banana', 'cherry']

JOIN TWO LISTS: There are several ways to join, or concatenate, two or more lists in Python.

1. **One of the easiest ways are by using the + operator.**

Example: Join two lists.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

Result: [‘a’, ’b’, ‘c’, 1, 2, 3]

2. **Another way to join two lists is by appending all the items from list2 into list1, one by one**.

Example: Append list2 into list1.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
for x in list2:
  list1.append(x)
print(list1)
```

Result: [‘a’, ’b’, ‘c’, 1, 2, 3]

3. **You can use the extend() method, which purpose is to add elements from one list to another list**.

Example: Use the extend() method to add list2 at the end of list1.

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

Result: [‘a’, ’b’, ‘c’, 1, 2, 3]

SORTING THE LISTS

1. **Sort List Alphanumerically:** List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

Example: Sort the list alphabetically.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Result: [‘banana’, ‘kiwi’, ‘mango’, ‘orange’, ‘pineapple’]

Example: Sort the list numerically.

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

Result: [23, 50, 65, 82,100]

2. **Sort Descending:** To sort descending, use the keyword argument reverse = True.

Example: Sort the list descending.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```
Result: [‘pineapple’,’orange’, ‘mango’, ‘kiwi’, ‘banana’]

Example: Sort the list descending.

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```
Result: [100, 82, 65, 50, 23]

3. **Case Insensitive Sort:** By default the sort() method is case sensitive, resulting in all ***capital letters being sorted after lower case letters.***

Example: Case sensitive sorting can give an unexpected result.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```
Result: [‘Kiwi’, ‘Orange’, ‘banana’, ‘cherry’]

4. **Luckily we can use built-in functions as key functions when sorting a list**: So if you want a case-insensitive sort function, use str.lower as a key function.

Example: Perform a case-insensitive sort of the list.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

Result: [‘banana’, ‘cherry’, ‘Kiwi’, ‘Orange’]

5. **Reverse Order:** What if you want to reverse the order of a list, regardless of the alphabet? The reverse() method reverses the current sorting order of the elements.

Example: Reverse the order of the list items.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Result: [‘cherry’, ‘Kiwi’, ‘Orange’, ‘banana’]

TUPLES: Tuples are used to store multiple items in a single variable. A tuple is a collection which is ordered and **unchangeable**. Tuples are written with round brackets.

Example: Create a Tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```
Result: (‘apple’, ‘banana’, ‘cherry’)

1. **Tuple Items:** Tuple items are ordered, unchangeable, and allow duplicate values. Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered: When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable: Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates: Since tuple are indexed, tuples can have items with the same value:

Example: Tuples allow duplicate values:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```
Result: ('apple', 'banana', 'cherry', 'apple', 'cherry')

2. **Tuple Length:** To determine how many items a tuple has, use the len() function:

Example: Print the number of items in the tuple

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```
Result: 3

3. **Tuple Items - Data Types:** Tuple items can be of any data type

Example: string, int and Boolean data types

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

4. A tuple can contain different data types

Example: A tuple with strings,integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

5. **type( ):** From Python's perspective, tuples are defined as objects with the data type 'tuple':

Example: What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```
Result: class 'tuple'

6. **The tuple() Constructor:** It is also possible to use the tuple() constructor to make a tuple.

Example: Using the tuple() method to make a tuple

```
thistuple = tuple(("apple", "banana", "cherry"))          # note the double round-brackets
print(thistuple)
```

Result: ('apple', 'banana', 'cherry')

ACCESS TUPLE ITEMS

1. **Access Tuple Items:** You can access tuple items by referring to the index number, inside square brackets:

Example: Print the second item in the tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```
Result: banana

Note: The first item has index 0.

2. **Negative Indexing:** Negative indexing means start from the end. -1 refers to the last item, -2 refers to the second last item etc.

Example: Print the last item of the tuple

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```
Result: cherry

3. **Range of Indexes:** You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

Example: Return the third, fourth, and fifth item

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```
Result: (‘cherry’, ‘orange’, ‘kiwi’)

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

4. **By leaving out the start value, the range will start at the first item**

Example: This example returns the items from the beginning to, but NOT included, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```
Result: (‘apple’, ’banana’, ‘cherry’, ‘orange’)

5. **By leaving out the end value, the range will go on to the end of the list**

Example: This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```
Result: (‘cherry’, ‘orange’, ‘kiwi’, ‘melon’, ‘mango’)

6. **Range of Negative Indexes:** Specify negative indexes if you want to start the search from the end of the tuple:

Example: This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Result: (‘orange’, ‘kiwi’, ‘melon’)

7. **Check if Item Exists:** To determine if a specified item is present in a tuple use the in keyword:

Example: Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Result: Yes, ‘apple’ is in the fruits tuple

Python - Update Tuples

1. **Change Tuple Values:** Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example: Convert the tuple into a list to be able to change it

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Result: (“apple”, “kiwi”, “cherry”)

2. **Add Items:** Once a tuple is created, you cannot add items to it.

Example: You cannot add items to a tuple

```
thistuple = ("apple", "banana", "cherry")
thistuple.append("orange") # This will raise an error
print(thistuple)
```

Result: ‘tuple’ object has no attribute ‘append’

Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example: Convert the tuple into a list, add "orange", and convert it back into a tuple

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Result: (‘apple’, ‘banana’, ‘cherry’, ‘orange’)

3. **Remove Items:** Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example: Convert the tuple into a list, remove "apple", and convert it back into a tuple

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Result: ('banana', 'cherry')

4. **Or you can delete the tuple completely**

Example: The del keyword can delete the tuple completely

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Loop through a Tuple

1. **Loop Through a Tuple:** You can loop through the tuple items by using a for loop.

Example: Iterate through the items and print the values

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Result: apple, banana, cherry

2. **Loop Through the Index Numbers:** You can also loop through the tuple items by referring to their index number. Use the range() and len() functions to create a suitable iterable.

Example: Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Result: apple, banana, cherry

3. **Using a While Loop:** You can loop through the list items by using a while loop. Use the len() function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Example: Print all items, using a while loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

4. **Join Two Tuples:** To join two or more tuples you can use the + operator:

Example: Join two tuples

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Result: (‘a’, ‘b’, ‘c’, 1, 2, 3)

5. **Multiply Tuples:** If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example: Multiply the fruits tuple by 2

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

Result: (‘apple’, ‘banana’, ‘cherry’, ‘apple’, ‘banana’, ‘cherry’)

Sets

Sets are used to store multiple items in a single variable. A set is a collection which is both *unordered* and *unindexed*. Sets are written with curly brackets.

Example: Create a Set

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

1. **Set Items:** Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered: Unordered means that the items in a set do not have a defined order.Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable: Sets are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can add new items.

Duplicates Not Allowed: Sets cannot have two items with the same value.

Example: Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

2. **Get the Length of a Set:** To determine how many items a set has, use the len() method.

Example: Get the number of items in a set

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

3. **Set Items - Data Types:** Set items can be of any data type:

Example: String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

Example: A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

4. **type():** From Python's perspective, sets are defined as objects with the data type 'set':

<class 'set'>
Example: What is the data type of a set?

```
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

5. **The set() Constructor:** It is also possible to use the set() constructor to make a set.

Example: Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

Access Items- Set

You cannot access items in a set by referring to an index or a key. You can loop through the set items using a for loop.

Example: Loop through the set, and print the values

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Result: cherry, apple,banana

1. You may ask if a specified value is present in a set, by using the in keyword.

Example: Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

Result: True

2. Change Items: Once a set is created, you cannot change its items, but you can add new items.

3. Add Items: To add one item to a set use the add() method.

Example: Add an item to a set, using the add() method

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Result:{‘cherry’, ‘apple’, ‘banana’, ‘orange’}

4. Remove Items: To remove an item in a set, use the remove(), or the discard() method.

Example: Remove "banana" by using the remove() method

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Result: {‘cherry’, ‘apple’}

Note: If the item to remove does not exist, remove() will raise an error.

5. Example: Remove "banana" by using the discard() method:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Result: {‘apple’, ‘cherry’}

Note: If the item to remove does not exist, discard() will **NOT** raise an error.

6. You can also use the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed. The return value of the pop() method is the removed item.

Example: Remove the last item by using the pop() method

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Result: cherry
{‘apple’, ‘banana’}

Note: Sets are *unordered*, so when using the pop() method, you do not know which item that gets removed.

7. Example: The del keyword will delete the set completely

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

Result: #this will raise an error because the set no longer exists

8. Loop Items: You can loop through the set items by using a for loop

Example: Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Result: cherry, banana, apple

Join Two Sets: There are several ways to join two or more sets in Python.

1. You can use the union() method that returns a new set containing all items from both sets.

Example: union() method returns a new set with all items from both sets

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

Result: {'b', 2, 1, 'a', 'c', 3}

2. You can use the update() method that inserts all the items from one set into another

Example: The update() method inserts the items in set2 into set1

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

Result: {1, 'a', 2, 'c', 3, 'b'}

Note: Both union() and update() will exclude any duplicate items.

Keep ONLY the Duplicates

3. The intersection() method will return a *new* set, that only contains the items that are present in both sets.

Example: Keep the items that exist in both set x, and set y

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)
```

Result:{'apple'}

Keep All, But NOT the Duplicates

4. The symmetric_difference() method will return a new set, that contains only the elements that are NOT present in both sets.

Example: Return a set that contains all items from both sets, except items that are present in both

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)
```

Result: {'google', 'banana', 'microsoft', 'cherry'}

Python Dictionaries: Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is unordered, changeable and does not allow duplicates. Dictionaries are written with curly brackets, and have keys and values:

Example: Create and print a dictionary

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

```
print(thisdict)
```
Result:{‘brand’:’Ford’,’model’:’Mustang’,’year’:1964}

1. **Dictionary Items:** Dictionary items are unordered, changeable, and does not allow duplicates. Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example: Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```
Result: Ford

Unordered: When we say that dictionaries are unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable: Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

***Duplicates Not Allowed*:** Dictionaries cannot have two items with the same key:

Example: Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```
Result: :{‘brand’:’Ford’,’model’:’Mustang’,’year’:2020}

2. **Dictionary Length:** To determine how many items a dictionary has, use the len() function:

Example: Print the number of items in the dictionary:

```
print(len(thisdict))
```
Result:3

3. **Dictionary Items - Data Types:** The values in dictionary items can be of any data type:

Example: String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

4. **type():** From Python's perspective, dictionaries are defined as objects with the data type 'dict'

Example: Print the data type of a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

<class ‘dict’>

Accessing Items-Dictionaries: You can access the items of a dictionary by referring to its key name, inside square brackets:

Example: Get the value of the "model" key

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
```

Result: Mustang

There is also a method called get() that will give you the same result:

Example: Get the value of the "model" key

```
x = thisdict.get("model")
```

1. Get Keys: The keys() method will return a list of all the keys in the dictionary.

Example: Get a list of the keys

```
x = thisdict.keys()
```

Result: dict_keys([‘brand’, ‘model’, ‘year’])

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example: Add a new item to the original dictionary, and see that the keys list gets updated as well

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.keys()
print(x) #before the change
car["color"] = "white"
print(x) #after the change
```

Result: dict_keys([‘brand’, ‘model’, ‘year’])
Dict_keys([‘brand’, ‘model’, ‘year’, ‘color’)]

2. Get Values: The values() method will return a list of all the values in the dictionary.

Example: Get a list of the values

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example: Add a new item to the original dictionary, and see that the keys list gets updated as well

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Result: dict_values([‘Ford’, ‘Mustang’, ‘1964])
dict_values([‘Ford’, ‘Mustang’, 2020])

3. **Get Items:** The items() method will return each item in a dictionary, as tuples in a list.

Example: Get a list of the key:value pairs

```
x = thisdict.items()
```

Result: dict_items([(‘brand’, ‘Ford’), (‘model’, ‘Mustang’), (‘year’, 1964)])

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example: Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

4. **Check if Key Exists:** To determine if a specified key is present in a dictionary use the in keyword:

Example: Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Dictionary Values: You can change the value of a specific item by referring to its key name:

Example: Change the "year" to 2018

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary: The update() method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs.

Example: Update the "year" of the car by using the update() method

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Adding Items: Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
Example: thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items: There are several methods to remove items from a dictionary.

Example: The pop()method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Example: The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example: The del keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Example: The del keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Example: The clear() method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Loop Through a Dictionary: You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example: Print all key names in the dictionary, one by one

```
thisdict =      {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
for x in thisdict:
  print(x)
```

Result: brand, model,year

Example: Print all *values* in the dictionary, one by one

```
for x in thisdict:
  print(thisdict[x])
```

Result: Ford, Mustang,1964

Example: You can also use the values() method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Result: Ford, Mustang,1964

Example: You can use the keys() method to return the keys of a dictionary

```
for x in thisdict.keys():
  print(x)
```

Result: brand, model, year

Example: Loop through both *keys* and *values*, by using the items() method:

```
for x, y in thisdict.items():
  print(x, y)
```

Result: brand Ford
model Mustang
year 1964

Copy a Dictionary

1. There are ways to make a copy, one way is to use the built-in Dictionary method copy().

Example: Make a copy of a dictionary with the copy() method

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

2. Another way to make a copy is to use the built-in function dict().

Example: Make a copy of a dictionary with the dict() function

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Nested Dictionaries: A dictionary can contain dictionaries, this is called nested dictionaries.

Example: Create a dictionary that contain three dictionaries

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
```

```
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

Example: Create three dictionaries, then create one dictionary that will contain the other three dictionaries

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

CHAPTER-5 (FUNCTIONS)

Python Functions: A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

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***.

1. **Defining a Function:** 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 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.

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
```

2. **Creating a Function:** In Python a function is defined using the def keyword. 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.

```
def my_function():
  print("Hello from a function")
```

3. **Calling a Function:** To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")
my_function()
```

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

4. **Arguments:** Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

5. **Parameters or Arguments?:** The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective: ***A parameter is the variable listed inside the parentheses in the function definition***.

An argument is the value that is sent to the function when it is called.

6. **Number of Arguments:** By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example: This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

Result: Emil Refsnes

If you try to call the function with 1 or 3 arguments, you will get an error:

Example: This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil")
```

Result: Error

Function Arguments

1. **Pass by reference vs value:** All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
```

```
   mylist.append([1,2,3,4]);
   print "Values inside the function: ", mylist
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. ***So, this would produce the following result*** −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

2. **Arbitrary Arguments, *args:** If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Example: If the number of arguments is unknown, add a * before the parameter name.

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Result: The youngest child is Linus

Arbitrary Arguments are often shortened to **args* in Python documentations.

3. **Keyword Arguments:** You can also send arguments with the *key = value* syntax. This way the order of the arguments does not matter.

Example:

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

4. **Arbitrary Keyword Arguments, **kwargs:** If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

Example: If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Result: His last name is Refsnes

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

5. **Default Parameter Value:** The following example shows how to use a default parameter value. If we call the function without argument, it uses the default value:

Example:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
```

Result: I am from Sweden
I am from India
I am from Norway

6. **Passing List as an Argument:** You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

7. **Return Values:** To let a function return a value, use the return statement:

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Result: 15, 25, 45

8. **The pass Statement:** function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

9. **Recursion:** Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Result: Recursion Example Results

1, 3, 6, 10, 15,21

Python Lambda: A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax: The syntax of *lambda* functions contains only a single statement as follows:

lambda [arg1 [,arg2,.....argn]]:expression

Syntax: lambda *arguments*: *expression*

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

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

```
Value of total:  30
Value of total:  40
```

The return Statement: 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.

All the above examples are not returning any value. You can return a value from a function as follows −

```
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum(10, 20 );
print "Outside the function : ", total
```

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

Inside the function: 30
Outside the function: 30

Example: Add 10 to argument a, and return the result

```
x = lambda a: a + 10
print(x(5))
```

Result: 15

Lambda functions can take any number of arguments:

Example: Multiply argument a with argument b and return the result:

```
x = lambda a, b: a * b
print(x(5, 6))
```

Result: 30

Example: Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Result: 13

a) Why Use Lambda Functions?: The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use function definition to make a function that always doubles the number you send in:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Result: 22

Or, use the same function definition to make a function that always *triples* the number you send in:

```
def myfunc(n):
  return lambda a : a * n
```

```
mytripler = myfunc(3)
print(mytripler(11))
```
Result: 33

Or, use the same function definition to make both functions, in the same program:

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```
Result: 22, 33

Use lambda functions when an anonymous function is required for a short period of time.

Scope of Variables

A variable is only available from inside the region it is created. This is called **scope**. Variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

1. **Global vs. Local variables:** Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;
# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

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

Inside the function local total: 30

Outside the function global total: 0

2. **Local Scope:** A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example: A variable created inside a function is available inside that function

```
def myfunc():
  x = 300
  print(x)
myfunc()
```

Result: 300

3. **Function Inside Function:** As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function.

Example: The local variable can be accessed from a function within the function

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()
myfunc()
```

Result: 300

4. **Global Scope:** A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example: A variable created outside of a function is global and can be used by anyone

```
x = 300
def myfunc():
  print(x)
myfunc()
print(x)
```

Result: 300, 300

5. **Naming Variables:** If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function)

Example: The function will print the local x, and then the code will print the global x

```
x = 300
def myfunc():
  x = 200
  print(x)
myfunc()
print(x)
```

Result: 200, 300

6. **Global Keyword:** If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

Example: If you use the global keyword, the variable belongs to the global scope

```
def myfunc():
  global x
  x = 300
myfunc()
print(x)
```

Result: 300

Also, use the global keyword if you want to make a change to a global variable inside a function.

Example: To change the value of a global variable inside a function, refer to the variable by using the global keyword

```
x = 300
def myfunc():
  global x
  x = 200
myfunc()
print(x)
```

Result: 200

CHAPTER-6 (MODULES)

Python Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

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.

Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

1 Create a Module: To create a module just save the code you want in a file with the file extension .py:

Example: Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Example: Here's an example of a simple module, support.py

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

2. Use a Module: Now we can use the module we just created, by using the import statement.

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.

Example: Import the module named mymodule, and call the greeting function:

```
import mymodule
mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

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.

3. 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.

4. 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.

5. Locating Modules: When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

6. The *PYTHONPATH* Variable: The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system −

set PYTHONPATH = c:\python20\lib;

And here is a typical PYTHONPATH from a UNIX system −

set PYTHONPATH = /usr/local/lib/python

Variables in Modules

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example: Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
```

```
  "country": "Norway"
}
```

Example: Import the module named mymodule, and access the person1 dictionary:

```
import mymodule
a = mymodule.person1["age"]
print(a)
```
Result: 36

1. **Re-naming a Module:** You can create an alias when you import a module, by using the as keyword:

Example: Create an alias for mymodule called mx:

```
import mymodule as mx
a = mx.person1["age"]
print(a)
```
Result: 36

2. **Built-in Modules:** There are several built-in modules in Python, which you can import whenever you like.

Example: Import and use the platform module:

```
import platform
x = platform.system()
print(x)
```
Result: Windows

3. **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.

   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.

4. **Using the dir() Function:** There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

Example: List all the defined names belonging to the platform module:

```
import platform
x = dir(platform)
print(x)
```

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
content = dir(math)
print content
```

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

['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']

Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

Note: The dir() function can be used on *all* modules, also the ones you create yourself.

5. **The *globals()* and *locals()* Functions:** The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

6. **The *reload()* Function:** When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this −

reload(module_name)

Here, *module_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following −

reload(hello)

7. **Import From Module:** You can choose to import only parts from a module, by using the from keyword.

Example: The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway" }
```

Example: Import only the person1 dictionary from the module:

```
from mymodule import person1
print (person1["age"])
```

When importing using the from keyword, do not use the module name when referring to elements in the module. Example: person1["age"], **not** ~~mymodule.person1["age"]~~

8. **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 & has following line of 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.

```
# 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 function 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.

9. **Python Dates:** A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

Example: Import the datetime module and display the current date

```
import datetime
x = datetime.datetime.now()
print(x)
```

Result: Displays current date & time

10. **Date Output:** When we execute the code from the example above the result is

Current date & time

The date contains year, month, day, hour, minute, second, and microsecond. The datetime module has many methods to return information about the date object.

Example: Return the year and name of weekday

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

Result: 2021, Saturday

11. **Creating Date Objects:** To create a date, we can use the datetime() class (constructor) of the datetime module. The datetime() class requires three parameters to create a date: year, month, day.

Example: Create a date object

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

12. **The strftime() Method:** The datetime object has a method for formatting date objects into readable strings. The method is called strftime(), and takes one parameter, format, to specify the format of the returned string.

Example: Display the name of the month

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

Result: June

CHAPTER-7 (EXCEPTIONS)

Python - Exception Handling

A python program terminates as soon as it encounters an unhandled error. These errors can be classified into two classes:

1. Syntax errors: Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
   If a>3
   Here a colon (:) is missing in the if statement.

2. Logical errors (Exceptions): Errors that occur at runtime (after passing the syntax test) are called Exceptions or Logical errors

Common exceptions:

S.N.	Exception	Cause of error
1.	AssertionError	Raised when an assert statement fails
2.	FileNotFoundError	To open a file (for reading) that does not exist
3.	ImportError	To import a module that does not exist
4.	MemoryError	Raised when an operation runs out of memory
5.	OverflowError	Raised when result of an arithmetic operation is too large
6.	SystemError	Raised when interpreter detects internal error
7.	ZeroDivisionError	To divide a number by zero

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

- Exception Handling
- Assertions

***Assertions in Python*:** An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program. The easiest way to think of an assertion is to liken it to a **raise-if** statement. An expression is tested, and if the result comes up false, an exception is raised. Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

The assert Statement: When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is: assert Expression[, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example: Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

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

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception? An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
The try block lets you test a block of code for errors.
The except block lets you handle the error.
The finally block lets you execute code, regardless of the result of the try- and except blocks.

Handling an exception: If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement:

Example: The try block will generate an exception, because x is not defined

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error:

Example: This statement will raise an error, because x is not defined:

```
print(x)
```

Handling Exception using try....except...else block

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Example: This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result − Written content in the file successfully

Example: This example tries to open a file where you do not have write permission, so it raises an exception −

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result − Error: can't find file or read data

The *except* Clause with Multiple Exceptions: You can also use the same *except* statement to handle multiple exceptions as follows −

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example: Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Result: Variable x is not defined

Else block: You can use the else keyword to define a block of code to be executed if no errors were raised:

Example: In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Result: Hello
Nothing went wrong

The try-finally clause: The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example:

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Result: Something went wrong
The 'try except' is finished

This can be useful to close objects and clean up resources:

Example: Try to open and write to a file that is not writable (i.e. read only file)

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

Result: Something went wrong when writing to the file

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the ***following result*** − Error: can't find file or read data

Same example can be written more clearly as follows −

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

Result: Going to close the file

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

Raise an exception:

As a Python developer you can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the raise keyword.

Example: Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

Result: Sorry, no numbers below zero

The raise keyword is used to raise an exception. You can define what kind of error to raise, and the text to print to the user.

Example: Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Result: TypeError: Only integers are allowed

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax: raise [Exception [, args [, traceback]]]

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

***Example*:** An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows –

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
```

```
  def __init__(self, arg):
    self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
  raise Networkerror("Bad hostname")
except Networkerror,e:
  print e.args
```

CHAPTER-8 (INPUT AND OUTPUT)

Reading Keyboard Input

Python allows for user input. That means we are able to ask the user for input. Python 3.6 uses the input() method. Python 2.7 uses the raw_input() method. The following example asks for the username, and when you entered the username, it gets printed on the screen.

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]
```

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?
```

String format()

The format() method allows you to format selected parts of a string. Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input? To control such values, add placeholders (curly brackets {}) in the text, and run the values through the format() method:

Example: Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

Result: The price is 49 dollars

You can add parameters inside the curly brackets to specify how to convert the value:

Example: Format the price to be displayed as a number with two decimals:

```
txt = "The price is {:.2f} dollars"
```

Result: The price is 49.00 dollars

Multiple Values

If you want to use more values, just add more values to the format() method and add more placeholders.

```
print(txt.format(price, itemno, count))
```

Example:

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Result: I want 3 pieces of item number 567 for 49.00 dollars.

Index Numbers

You can use index numbers (a number inside the curly brackets {0}) to be sure the values are placed in the correct placeholders.

Example:

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Also, if you want to refer to the same value more than once, use the index number.

Example:

```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

Named Indexes

You can also use named indexes by entering a name inside the curly brackets {carname} but then you must use names when you pass the parameter values txt.format(carname = "Ford").

Example:

```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

Opening and Closing 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).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

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

S.N.	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
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
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
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

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 −

SN.	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()

Example:

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
# Close opened file
fo.close()
```

This produces the following result −

Name of the file: foo.txt

The *read()* Method: The *read()* method reads a string from an open file. It is important to note that Python strings can have binary data apart from text data.

Syntax: fileObject.read([count])

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if *count* is missing, then it tries to read as much as possible, maybe until the end of file.

Example: Let's take a file *foo.txt,* which we created as:

Python is a great language.

Yeah its great!!

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opened file
fo.close()
```

This produces the following result −

Read String is : Python is

Open a File on the Server: Assume we have the following file, located in the same folder as Python:

demofile.txt

Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!

To open the file, use the built-in `open()` function. The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example: Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File: By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example: Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines: You can return one line by using the `readline()` method:

Example: Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling `readline()` two times, you can read the two first lines:

Example: Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example: Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files: It is a good practice to always close the file when you are done with it.

Example: Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

The *write()* Method: The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string −

Syntax: fileObject.write(string). Here, passed parameter is the content to be written into the opened file.

```
# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n")

# Close opened file
fo.close()
```

The above method would create *foo.txt* file and would write given content in that file and finally it would close that file. If you would open this file, it would have following content.

```
Python is a great language.
Yeah its great!!
```

File Positions: The *tell()* method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(offset[, from])* method changes the current file position. The *offset* argument indicates the number of bytes to be moved. The *from* argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example: Let us take a file *foo.txt*, which we created above.

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print "Read String is : ", str

# Check current position
position = fo.tell()
print "Current file position : ", position

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10)
print "Again read String is : ", str
# Close opend file
fo.close()
```

This produces the following result −

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files: Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files. To use this module you need to import it first and then you can call any related functions.

The rename() Method: The *rename()* method takes two arguments, the current filename and the new filename.

Syntax: os.rename(current_file_name, new_file_name)

Example: Following is the example to rename an existing file *test1.txt* −

```
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The remove() Method: You can use the *remove()* method to delete files by supplying the name of the file to be deleted as the argument.

Syntax: os.remove(file_name)

Example: Following is the example to delete an existing file *test2.txt* −

```
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python: All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The mkdir() Method: You can use the *mkdir()* method of the **os** module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

Syntax: os.mkdir("newdir")

Example: Following is the example to create a directory *test* in the current directory −

```
import os

# Create a directory "test"
os.mkdir("test")
```

The chdir() Method: You can use the *chdir()* method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax: os.chdir("newdir")

Example: Following is the example to go into "/home/newdir" directory −

```
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

The getcwd() Method: The *getcwd()* method displays the current working directory.

Syntax: os.getcwd()

Example: Following is the example to give current directory −

```
import os

# This would give location of the current directory
os.getcwd()
```

The rmdir() Method: The *rmdir()* method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should be removed.

Syntax: os.rmdir('dirname')

Example: Following is the example to remove "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
import os

# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

CHAPTER-9 (CLASSES IN PYTHON)

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy.

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 behaviour 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.

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(empl, '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>}
```

Class Inheritance:

Instead of starting from scratch, you can create a class by deriving it from a 86re-existing 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
```

Example

```
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
      print "Calling parent constructor"

   def parentMethod(self):
      print 'Calling parent method'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "Parent attribute :", Parent.parentAttr
```

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"

   def childMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

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

Calling child constructor; Calling child method; Calling parent method; Parent attribute : 200

Similar way, you can drive a class from multiple parent classes as follows −

```
class A:        # define your class A
.....
class B:        # define your class B
.....
class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

Overriding Methods:

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

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

```
Calling child method
```

Overloading Operators:

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation −

Example:

```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

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

```
Vector(7,8)
```

Data Hiding:

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

Example:

```
class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.__secretCount
```

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

```
1
2
Traceback (most recent call last):
   File "test.py", line 12, in <module>
      print counter.__secretCount
```

AttributeError: JustCounter instance has no attribute '__secretCount'

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you −

```
.....................
print counter._JustCounter__secretCount
```

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

```
1
2
2
```

CHAPTER-10 (REGULAR EXPRESSIONS)

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The Python module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The *match* Function: This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

re.match(pattern, string, flags=0)

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**Pattern:** This is the regular expression to be matched.
2	**String:** This is the string, which would be searched to match the pattern at the beginning of string.
3	**Flags:** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	**group(num=0):** This method returns entire match (or specific subgroup num)
2	**groups():** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example:

```
import re
line = "Cats are smarter than dogs"
matchObj = re.match(r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
```

```
  print "No match!!"
```

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

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function: This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**Pattern:** This is the regular expression to be matched.
2	**String:** This is the string, which would be searched to match the pattern anywhere in the string.
3	**Flags:** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description
1	**group(num=0):** This method returns entire match (or specific subgroup num)
2	**groups():** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example:

```
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print "searchObj.group() : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
```

```
else:
   print "Nothing found!!"
```

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

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

Matching Versus Searching: Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default). E.g.

```
import re
line = "Cats are smarter than dogs";
matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print "search --> searchObj.group() : ", searchObj.group()
else:
   print "Nothing found!!"
```

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

```
No match!!
search --> searchObj.group() :  dogs
```

Search and Replace: One of the most important **re** methods that use regular expressions is **sub**.

Syntax: re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string. E.g.

```
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

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

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Sr.No.	Modifier & Description
1	**re.I:** Performs case-insensitive matching.
2	**re.L:** Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M:** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S:** Makes a period (dot) match any character, including a newline.
5	**re.U:** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	**re.X:** Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns: Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash. Following table lists the regular expression syntax that is available in Python −

Sr.No.	Pattern & Description
1	^ Matches beginning of line.
2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets

6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.
8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}** Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.
13	**(re)** Groups regular expressions and remembers matched text.
14	**(?#...)** Comment.

Regular Expression Examples:

Literal characters:

Sr.No.	Example & Description
1	**Python** Match "python".

Character classes

Sr.No.	Example & Description
1	**[Pp]ython:** Match "Python" or "python"
2	**rub[ye]:** Match "ruby" or "rube"
3	**[aeiou]:** Match any one lowercase vowel

4	**[0-9]:** Match any digit; same as [0123456789]
5	**[a-z]:** Match any lowercase ASCII letter
6	**[A-Z]:** Match any uppercase ASCII letter
7	**[a-zA-Z0-9]:** Match any of the above
8	**[^aeiou]:** Match anything other than a lowercase vowel
9	**[^0-9]:** Match anything other than a digit

Special Character Classes:

Sr.No.	Example & Description
1	**.** Match any character except newline
2	**\d** Match a digit: [0-9]
3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description

1	**ruby?** Match "rub" or "ruby": the y is optional
2	**ruby*** Match "rub" plus 0 or more ys
3	**ruby+** Match "rub" plus 1 or more ys
4	**\d{3}** Match exactly 3 digits
5	**\d{3,}** Match 3 or more digits
6	**\d{3,5}** Match 3, 4, or 5 digits

Nongreedy repetition: This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	**<.*>** Greedy repetition: matches "<python>perl>"
2	**<.*?>** Nongreedy: matches "<python>" in "<python>perl>"

Alternatives

Sr.No.	Example & Description
1	**python|perl:** Match "python" or "perl"
2	**rub(y|le)):** Match "ruby" or "ruble"
3	**Python(!+|\?):** "Python" followed by one or more ! or one ?

Anchors: This needs to specify match position.

Sr.No.	Example & Description

1	**^Python:** Match "Python" at the start of a string or internal line
2	**Python$:** Match "Python" at the end of a string or line
3	**\APython:** Match "Python" at the start of a string
4	**Python\Z:** Match "Python" at the end of a string
5	**\bPython\b:** Match "Python" at a word boundary
6	**\brub\B:** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	**Python(?=!):** Match "Python", if followed by an exclamation point.
8	**Python(?!!):** Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	**R(?#comment):** Matches "R". All the rest is a comment
2	**R(?i)uby:** Case-insensitive while matching "uby"
3	**R(?i:uby):** Same as above
4	**rub(?:y|le)):** Group only without creating \1 backreference