University of Technology

Computer Science Department.

الجامعـــــة التكنولوجيـــــة

قسـم علوم الحاسوب

PYTHON LANGUAGE & NLP

AI -1'ST COURSE

2^{nd} *Class*

Mustafa Jasim Hadi

2017-2018

PYTHON OVERVIEW

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

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

History of Python

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

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

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

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

Python Features

Python's features include:

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:**Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

- 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 provides very high-level dynamic data types and supports dynamic type checking.
- IT supports automatic garbage collection.

Libraries for Natural Language & Text Processing (NLTK Tool)
NLTK : is an Open source Python modules, linguistic data and documentation for research and development in natural language processing and text analytics.

Steps to install Python on Windows machine.
1- Open a Web browser and go to <http://www.python.org/download/>
2- Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.
3- 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.
4- 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.

Setting path at Windows
To add the Python directory to the path for a particular session in Windows:
At the command prompt: type path "%path%;C:\Python" and press Enter.
Note: "C:\Python" is the path of the Python directory.

Running Python
There are three different ways to run Python:
1-Interactive Interpreter: starting Python from DOS, or any other system that provides a command-line interpreter or shell window.
2- Script from the Command-line: A Python script can be executed at command line by invoking the interpreter on your application.
3- Integrated Development Environment: Run Python from a Graphical User Interface (GUI) environment, if you have a GUI application on your system that supports Python. PythonWin is the first Windows interface for Python and is an IDE with a GUI.

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following 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*****!** " **;**. However in Python version 2.4.3, this produces the following result:

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py 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!
```

Let us try another way to execute a Python script. Here is the modified test.py file −

```
#!/usr/bin/python

print "Hello, Python!"
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows −

```
$ chmod +x test.py     # This is to make file executable
$./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.

And	exec	Not
Assert	finally	or
Break	for	pass
Class	from	print
Continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

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. The following example has various statement blocks −

Note: Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

```
#!/usr/bin/python

import sys

try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print "There was an error writing to", file_name
  sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
  print "Next time please enter something"
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print "There was an error reading file"
  sys.exit()
file_text = file.read()
file.close()
print file_text
```

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']
```

Quotation in Python

Python accepts single ', double " and triple ''' 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.

```
#!/usr/bin/python

# 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
```

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.
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action −

```
#!/usr/bin/python

raw_input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

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
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h −

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone

Codes to get the Python version you are using:
import sys; print (sys.version); print (sys.version_info)

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 −

```
#!/usr/bin/python

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
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

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

Python has five standard data types −

- Numbers
- String

- List
- Tuple
- Dictionary

Python Numbers

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

```
var1 = 1
var2 = 10
```

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

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

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

```
del var
del var_a, var_b
```

Python supports four different numerical types −

- int *signedintegers*
- long *longintegers, theycanalsoberepresentedinoctalandhexadecimal*
- float *floatingpointrealvalues*
- complex *complexnumbers*

Examples

Here are some examples of numbers −

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

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

Python Strings

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

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

```
#!/usr/bin/python

str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result −

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

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

The 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 −

```
#!/usr/bin/python

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', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

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 −

```
#!/usr/bin/python

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

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

This produce the following result −

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 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 −

```
#!/usr/bin/python

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
```

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 −

```
#!/usr/bin/python

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.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Function	Description
int*x*[, *base*]	Converts x to an integer. base specifies the base if x is a string.
long*x*[, *base*]	Converts x to a long integer. base specifies the base if x is a string.
float*x*	Converts x to a floating-point number.
complex*real*[, *imag*]	Creates a complex number.
str*x*	Converts object x to a string representation.
repr*x*	Converts object x to an expression string.
eval*str*	Evaluates a string and returns an object.
tuple*s*	Converts s to a tuple.
list*s*	Converts s to a list.
set*s*	Converts s to a set.
dict*d*	Creates a dictionary. d must be a sequence of *key*, *value* tuples.
frozenset*s*	Converts s to a frozen set.
chr*x*	Converts an integer to a character.
unichr*x*	Converts an integer to a Unicode character.

ord*x*	Converts a single character to its integer value.
hex*x*	Converts an integer to a hexadecimal string.
oct*x*	Converts an integer to an octal string.

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

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

Let us have a look on all operators one by one.

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
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

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, then −

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.	
<>	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.

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 / ac /= 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	Performs exponential *power* calculation on operators and assign value to the left	c **= a is equivalent to c = c ** a

AND	operand	
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

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*00001100
| Binary OR	It copies a bit if it exists in either operand.	$a \mid b$ = 61 *means*00111101
^ Binary XOR	It copies the bit if it is set in one operand but not both.	a^b = 49 *means*00110001
~ 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 << = 240 *means*11110000
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 *means*00001111

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Used to reverse the logical state of its operand.

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.

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$y$.

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

|

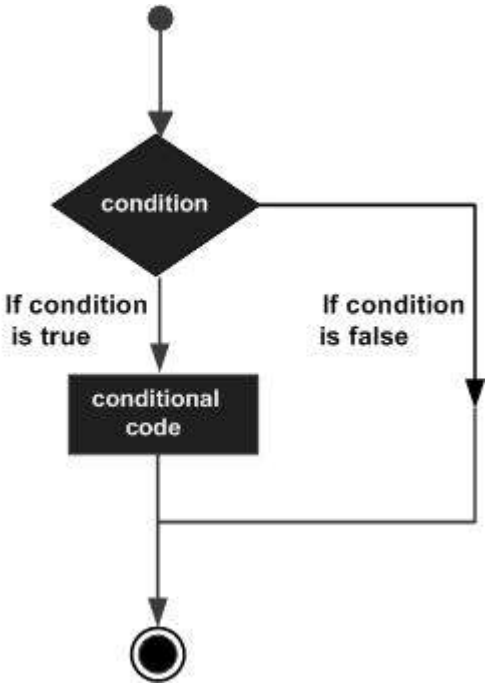
Operator	Description
**	Exponentiation $raisetothepower$
~ + -	Ccomplement, unary plus and minus method names for the last two are +@ and -@
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators

in not in	Membership operators
not or and	Logical operators

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

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

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



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

Python programming language provides following types of decision making statements.

Statement	Description
if statements	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statements	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statements.

Let us go through each decision making briefly −

Single Statement Suites

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

Here is an example of a **one-line if** clause −

```
#!/usr/bin/python

var = 100

if ( var  == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

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

```
Value of expression is 100
Good bye!
```

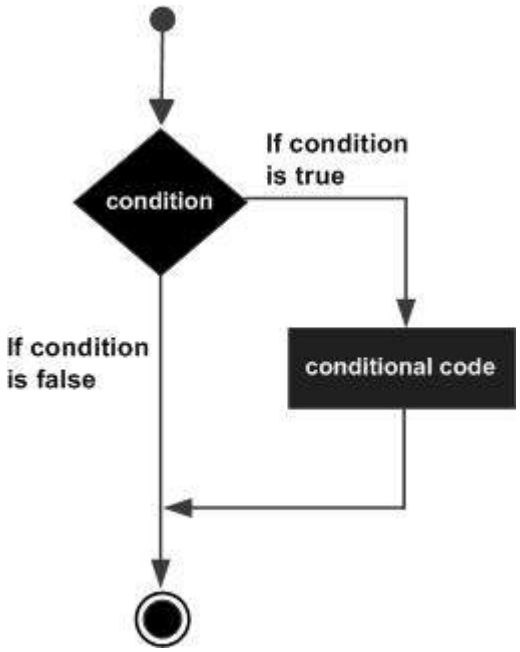
It is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statements inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statements is executed.

Flow Diagram



Example

```
#!/usr/bin/python

var1 = 100
if var1:
   print "1 - Got a true expression value"
   print var1

var2 = 0
if var2:
   print "2 - Got a true expression value"
   print var2
print "Good bye!"
```

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

```
1 - Got a true expression value
100
Good bye!
```

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

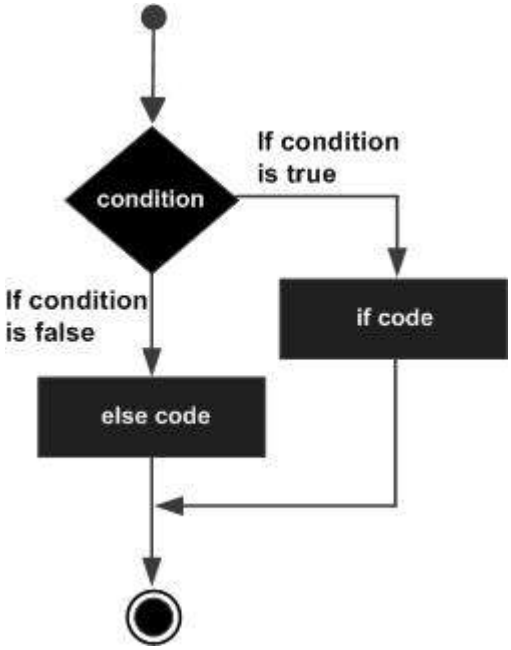
The *else* statement is an optional statement and there could be at most only one **else** statement following **if** .

Syntax

The syntax of the *if...else* statement is −

```
if expression:
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python

var1 = 100
if var1:
   print "1 - Got a true expression value"
   print var1
else:
   print "1 - Got a false expression value"
   print var1

var2 = 0
if var2:
   print "2 - Got a true expression value"
   print var2
else:
   print "2 - Got a false expression value"
   print var2

print "Good bye!"
```

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

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

The *elif* Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows −

Example

```
#!/usr/bin/python

var = 100
if var == 200:
   print "1 - Got a true expression value"
   print var
elif var == 150:
   print "2 - Got a true expression value"
   print var
elif var == 100:
   print "3 - Got a true expression value"
   print var
else:
   print "4 - Got a false expression value"
   print var

print "Good bye!"
```

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

```
3 - Got a true expression value
100
Good bye!
```

PYTHON NESTED IF STATEMENTS

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax:

The syntax of the nested *if...elif...else* construct may be:

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example:

```
#!/usr/bin/python

var = 100
if var < 200:
   print "Expression value is less than 200"
   if var == 150:
      print "Which is 150"
   elif var == 100:
      print "Which is 100"
   elif var == 50:
      print "Which is 50"
elif var < 50:
   print "Expression value is less than 50"
else:
   print "Could not find true expression"

print "Good bye!"
```

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

```
Expression value is less than 200
Which is 100
Good bye!
```

Examples Sheet #1 (Python Decision Making)

Example1: Python program to check if a number is positive, negative or zero.

In this example, you will learn to check whether a number entered by the user is positive, negative or zero. This problem is solved using if...elif...else and nested if...else statement.

1- Using if...elif...else

```
num = float(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

2- Using Nested if

```
num = float(input("Enter a number: "))
if num >= 0:
   if num == 0:
      print("Zero")
   else:
      print("Positive number")
else:
   print("Negative number")
```

The output of both programs will be same.

Output 1

```
Enter a number: 2
Positive number
```

Output 2

```
Enter a number: 0
Zero
```

Output 3

```
Enter a number: -1
Negative number
```

Example2: Python Program to Check if a Number is Odd or Even

In this example, you will learn to check whether a number entered by the user is even or odd.

```
# Python program to check if the input number is odd or even.
# A number is even if division by 2 give a remainder of 0.
# If remainder is 1, it is odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Output 1

```
Enter a number: 43
43 is Odd
```

Output 2

```
Enter a number: 18
18 is Even
```

Example3: Python Program to Check Leap Year

In this program, you will learn to check whether a year is leap year or not. We will use nested if...else to solve this problem.

A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400. For example,

```
2017 is not a leap year
1900 is a not leap year
2012 is a leap year
2000 is a leap year
```

```
# Python program to check if the input year is a leap year or not
year = 2000
# To get year (integer input) from the user
# year = int(input("Enter a year: "))
if (year % 4) == 0:
   if (year % 100) == 0:
      if (year % 400) == 0:
         print("{0} is a leap year".format(year))
      else:
         print("{0} is not a leap year".format(year))
   else:
      print("{0} is a leap year".format(year))
else:
```

```
  print("{0} is not a leap year".format(year))
```

Output

```
2000 is a leap year
```

You can change the value of `year` in the source code and run it again to test this program.

Example4: Python Program to Check Prime Number

Example to check whether an integer is a prime number or not using for loop and if...else statement. If the number is not prime, it's explained in output why it is not a prime number.

```
# Python program to check if the input number is prime or not
num = 407
# take input from the user
# num = int(input("Enter a number: "))
# prime numbers are greater than 1
if num > 1:
   # check for factors
   for i in range(2,num):
       if (num % i) == 0:
           print(num,"is not a prime number")
           print(i,"times",num//i,"is",num)
           break
   else:
       print(num,"is a prime number")
# if input number is less than
# or equal to 1, it is not prime
else:
   print(num,"is not a prime number")
```

Output

```
407 is not a prime number
11 times 37 is 407
```

You can change the value of variable *num* in the above source code and test for other integers (if you want).

H.W1: Python Program to Print all Prime Numbers in an Interval, For Example:

```
The prime numbers between 900 and 1000 are:
907
911
919
929
937
941
947
953
967
971
977
983
991
997
```

H.W2: What is the output of the following code?

```
if None:
   print(“Hello”)
```

Choose one: (False, Hello, Nothing will be printed, Syntax error)

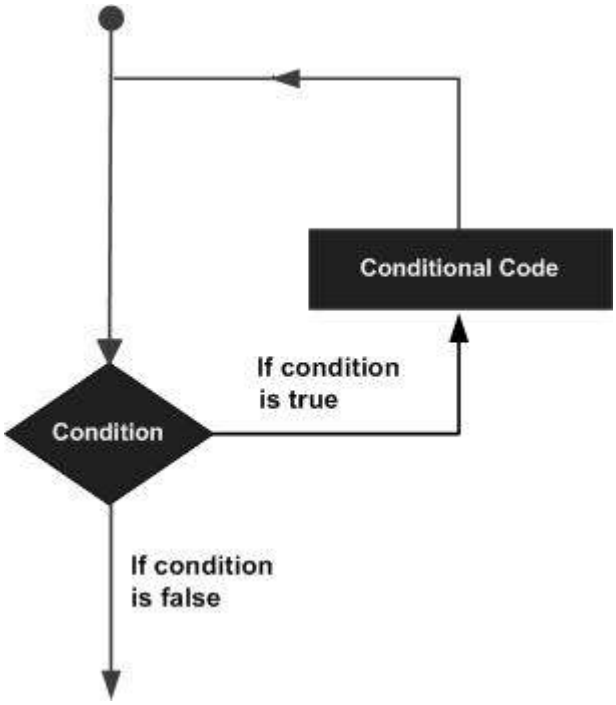
HW3.The if...elif...else executes only one block of code among several blocks.

Choose one:(True, False, It depends on expression used, here is no elif statement in Python).

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.

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

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



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

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

Control Statement	Description

break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly −

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

Syntax

The syntax of a **while** loop in Python programming language is −

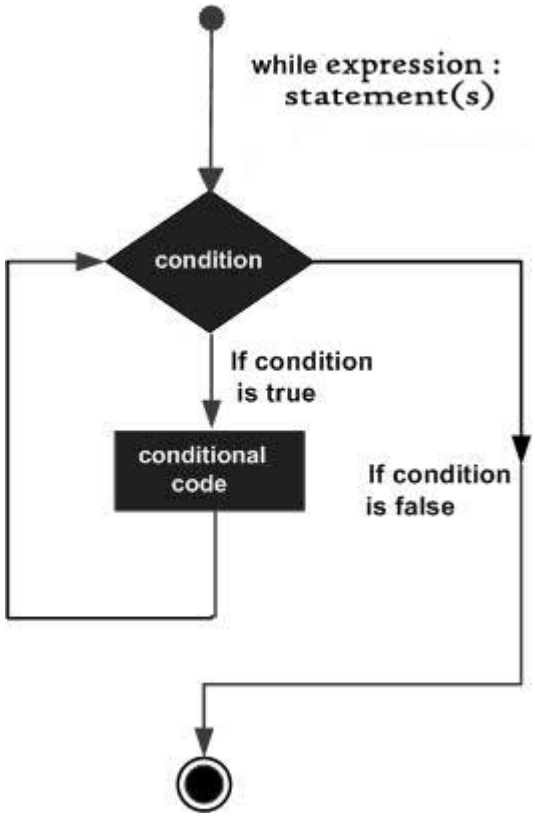
```
while expression:
   statement(s)
```

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.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#!/usr/bin/python

count = 0
while (count < 9):
   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
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. 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.

```
#!/usr/bin/python

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  :29
You entered:  29
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 Loops

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.
- 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 5, otherwise else statement gets executed.p>

```
#!/usr/bin/python

count = 0
while count < 5:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```

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

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

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 −

```
#!/usr/bin/python

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

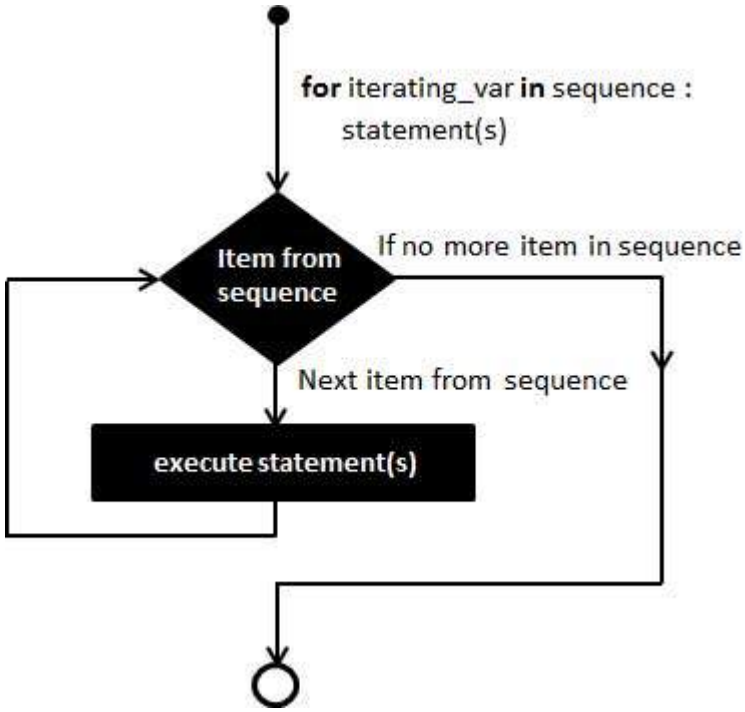
It has the ability to iterate over the items of any sequence, such as a list or a string.

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



Example

```
#!/usr/bin/python

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 −

```
#!/usr/bin/python

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 Loops

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.
- 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 for statement that searches for prime numbers from 10 through 20.

```
#!/usr/bin/python

for num in range(10,20):  #to iterate between 10 to 20
   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'
```

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
18 equals 2 * 9
19 is a prime number
```

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

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

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

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
#!/usr/bin/python

i = 2
while(i < 100):
   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
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
```

```
79 is prime
83 is prime
89 is prime
97 is prime
Good bye!
```

Examples Sheet #2 (Python Loops)

Example1: Use While loop to print numbers between 0 and 5:

```
# Solution1
count = 0
while count <= 5:
    print(count)
    count += 1  # This is the same as count = count + 1

# Solution2
# Use break statement with While loop to print out 0,1,2,3,4,5.
count = 0
while True:
    print(count)
    count += 1
    if count > 5:
        break

# Solution3
# Prints out 0,1,2,3,4,5 and then it prints "count value reached 6"
count=0
while(count<=5):
    print(count)
    count +=1
else:
    print("count value reached %d" %(count))
```

Example2: Use While loop to sum numbers from 1 to n.

```
# Program to add natural numbers up to sum = 1+2+3+...+n
n = 10 # It can be entered by user using: n = int(input("Enter n: "))
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1    # update counter

print "The sum is", sum
#Output:
#Enter n: 10
#The sum is 55
```

Example3: Use of Else statement with While loop to print "Inside loop" three times and print "Inside else" once After getting out of from the loop :

```
# Example to illustrate
# the use of else statement
# with the while loop
counter = 0
while counter < 3:
    print("Inside loop")
    counter = counter + 1
```

```
else:
    print("Inside else")
#Output:
#Inside loop
#Inside loop
#Inside loop
#Inside else
```

Example 4: use For loop to print numbers from 1 to 5:

```
for i in range(1, 10):
   if(i%6==0):
      break
   print(i)
else:
   print("this is not printed because For loop is terminated because of break but not due to fail in
condition")
```

Example 5: use For loop to print odd numbers between 1 to 10:

```
#Solution
for x in range(10):
   # Check if x is even
   if x % 2 == 0:
      continue
   print(x)
```

Example 6: Write a Python program that prints all the numbers from 0 to 6 except 3 and 6. Note : Use 'continue' statement, the output is 0 1 2 4 5.

```
#Solution
for x in range(6):
   if (x == 3 or x==6):
      continue
   print(x,end=' ')
print("\n")
```

Example 7: Write a Python program that accepts a string and calculate the number of digits and letters. Let the string is "Python 7.2", the output is:
Letters 6
Digits 2

```
#Solution
s = input("Input a string")
d=l=0
for c in s:
   if c.isdigit():
      d=d+1
   elif c.isalpha():
      l=l+1
   else:
      pass
print("Letters", l)
```

```
print("Digits", d)
```

Example 8: Write a Python program to construct the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

```
#Solution
n=5;
for i in range(n):
   for j in range(i):
      print ('* ', end="")
   print('')

for i in range(n,0,-1):
   for j in range(i):
      print('* ', end="")
   print('')
```

Example 9: Write a Python program to print alphabet pattern 'A'.

```
    * * *
  *       *
  *       *
  * * * * *
  *       *
  *       *
  *       *
```

```
#Solution
items = []
result_str="";
for row in range(0,7):
   for column in range(0,7):
      if (((column == 1 or column == 5) and row != 0) or ((row == 0 or row == 3) and (column > 1 and column
< 5))):
         result_str=result_str+"*"
      else:
         result_str=result_str+" "
   result_str=result_str+"\n"
print(result_str);
```

Example 10: Write a Python program to check the validity of password input by users. The validations are:

1. At least 1 letter between [a-z] and 1 letter between [A-Z].

2. At least 1 number between [0-9].
3. At least 1 character from [$#@].
4. Minimum length 6 characters.
5. Maximum length 16 characters.

```
import re
p= input("Input your password")
x = True
while x:
   if (len(p)<6 or len(p)>12):
      break
   elif not re.search("[a-z]",p):
      break
   elif not re.search("[0-9]",p):
      break
   elif not re.search("[A-Z]",p):
      break
   elif not re.search("[$#@]",p):
      break
   elif re.search("\s",p):
      break
   else:
      print("Valid Password")
      x=False
      break
if x:
   print("Not a Valid Password")
```

H.W1: What is the output of the following code?

Choose one: (2 1, 2 0, [2,1], [2,0]).

```
for i in [1, 0]:  print(i+1)
```

H.W2: In Python, for and while loop can have optional else statement?

Choose one:

1. **Only for loop can have optional else statement.**
2. **Only while loop can have optional else statement.**
3. **Both loops can have optional else statement.**
4. **Loops cannot have else statement in Python.**

H.W3: What is the output of the following code?

Choose one: (0,10,4,None).

```
i = sum = 0
while i <= 4:
   sum += i
   i = i+1
print(sum)
```

H.W4: What is the output of the following code?

Choose one:(4 is printed once, 4 is printed four times, 4 is printed infinitely until program closes, Syntax error).

```
while 4 == 4:  print('4')
```

H.W5: Is it better to use for loop instead of while if you are iterating through a sequence (like: list)?

Choose one:

1. **No, it's better to use while loop.**
2. **Yes, for loop is more pythonic choice.**
3. **No, you cannot iterate through a sequence using while loop.**
4. **No, you cannot iterate through a sequence using loops.**

H.W6: Which of the following statement is true?

Choose one:

1. **The break statement terminates the loop containing it.**
2. **The continue statement is used to skip the rest of the code inside the loop.**
3. **The break and continue statements are almost always used with if, if...else and if...elif...else statements.**
4. **All of the above.**

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example −

```
var1 = 1
var2 = 10
```

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

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

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

```
del var
del var_a, var_b
```

Python supports four different numerical types −

- **int** *signedintegers*: They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
- **long** *longintegers*: Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
- **float** *floatingpointrealvalues* : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).
- **complex** *complexnumbers* : are of the form a + bJ, where a and b are floats and J *orj* represents the square root of -1 *whichisanimaginarynumber*. The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Examples

Here are some examples of numbers

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

- Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int***x* to convert x to a plain integer.
- Type **long***x* to convert x to a long integer.
- Type **float***x* to convert x to a floating-point number.
- Type **complex***x* to convert x to a complex number with real part x and imaginary part zero.
- Type **complex***x, y* to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions

Python includes following functions that perform mathematical calculations.

Function	Returns *description*
abs*x*	The absolute value of x: the *positive* distance between x and zero.
ceil*x*	The ceiling of x: the smallest integer not less than x
cmp*x, y*	-1 if x < y, 0 if x == y, or 1 if x > y
exp*x*	The exponential of x: e^x
fabs*x*	The absolute value of x.
floor*x*	The floor of x: the largest integer not greater than x
log*x*	The natural logarithm of x, for x> 0
log10*x*	The base-10 logarithm of x for x> 0 .
max*x1, x2, . . .*	The largest of its arguments: the value closest to positive infinity
min*x1, x2, . . .*	The smallest of its arguments: the value closest to negative infinity
modf*x*	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

pow*x, y*	The value of x**y.
round*x*[, *n*]	**x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round0.5 is 1.0 and round−0.5 is -1.0.
sqrt*x*	The square root of x for x > 0

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

Function	Description
choice*seq*	A random item from a list, tuple, or string.
randrange [*start*,]*stop*[, *step*]	A randomly selected element from range*start*, *stop*, *step*
random	A random float r, such that 0 is less than or equal to r and r is less than 1
seed[*x*]	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle*lst*	Randomizes the items of a list in place. Returns None.
uniform*x, y*	A random float r, such that x is less than or equal to r and r is less than y

Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

Function	Description
acos*x*	Return the arc cosine of x, in radians.
asin*x*	Return the arc sine of x, in radians.
atan*x*	Return the arc tangent of x, in radians.
	Return atan*y*/*x*, in radians.

atan2y, x	
cos$x$	Return the cosine of x radians.
hypotx, y	Return the Euclidean norm, sqrt$x * x + y * y$.
sin$x$	Return the sine of x radians.
tan$x$	Return the tangent of x radians.
degrees$x$	Converts angle x from radians to degrees.
radians$x$	Converts angle x from degrees to radians.

Mathematical Constants

The module also defines two mathematical constants −

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.

Examples Sheet #3 (Python Numbers)

Example1: show the usage of abs() method by python codes.

```
print "abs(-45) : ", abs(-45)
print "abs(100.12) : ", abs(100.12)
print "abs(119L) : ", abs(119L)
```

When we run above program, it produces following result:

```
abs(-45) :  45
abs(100.12) :  100.12
abs(119L) :  119
```

Example2: show the usage of ceil() method by python codes.

```
import math   # This will import math module
print "math.ceil(-45.17) : ", math.ceil(-45.17)
print "math.ceil(100.12) : ", math.ceil(100.12)
print "math.ceil(100.72) : ", math.ceil(100.72)
print "math.ceil(119L) : ", math.ceil(119L)
print "math.ceil(math.pi) : ", math.ceil(math.pi)
```

When we run above program, it produces following result:

```
math.ceil(-45.17) :  -45.0
math.ceil(100.12) :  101.0
math.ceil(100.72) :  101.0
math.ceil(119L) :  119.0
math.ceil(math.pi) : 4.0
```

Example3: show the usage of cmp() method by python codes.

```
print "cmp(80, 100) : ", cmp(80, 100)
print "cmp(180, 100) : ", cmp(180, 100)
print "cmp(-80, 100) : ", cmp(-80, 100)
print "cmp(80, -100) : ", cmp(80, -100)
```

When we run above program, it produces following result:

```
cmp(80, 100) :  -1
cmp(180, 100) :  1
cmp(-80, 100) :  -1
cmp(80, -100) :  1
```

Example4: show the usage of exp () method by python codes.

```
import math   # This will import math module

print "math.exp(-45.17) : ", math.exp(-45.17)
print "math.exp(100.12) : ", math.exp(100.12)
print "math.exp(100.72) : ", math.exp(100.72)
print "math.exp(119L) : ", math.exp(119L)
print "math.exp(math.pi) : ", math.exp(math.pi)
```

When we run above program, it produces following result:

```
math.exp(-45.17) :  2.41500621326e-20
```

```
math.exp(100.12) :  3.03084361407e+43
math.exp(100.72) :  5.52255713025e+43
math.exp(119L) :  4.7978133273e+51
math.exp(math.pi) :  23.1406926328
```

Example5: show the usage of fabs() method by python codes.

```
import math   # This will import math module

print "math.fabs(-45.17) : ", math.fabs(-45.17)
print "math.fabs(100.12) : ", math.fabs(100.12)
print "math.fabs(100.72) : ", math.fabs(100.72)
print "math.fabs(119L) : ", math.fabs(119L)
print "math.fabs(math.pi) : ", math.fabs(math.pi)
```

When we run above program, it produces following result:

```
math.fabs(-45.17) :  45.17
math.fabs(100.12) :  100.12
math.fabs(100.72) :  100.72
math.fabs(119L) :  119.0
math.fabs(math.pi) :  3.14159265359
```

Example6: show the usage of floor() method by python codes.

```
import math   # This will import math module

print "math.floor(-45.17) : ", math.floor(-45.17)
print "math.floor(100.12) : ", math.floor(100.12)
print "math.floor(100.72) : ", math.floor(100.72)
print "math.floor(119L) : ", math.floor(119L)
print "math.floor(math.pi) : ", math.floor(math.pi)
```

When we run above program, it produces following result:

```
math.floor(-45.17) :  -46.0
math.floor(100.12) :  100.0
math.floor(100.72) :  100.0
math.floor(119L) :  119.0
math.floor(math.pi) :  3.0
```

Example7: show the usage of log() method by python codes.

```
import math   # This will import math module

print "math.log(100.12) : ", math.log(100.12)
print "math.log(100.72) : ", math.log(100.72)
print "math.log(119L) : ", math.log(119L)
print "math.log(math.pi) : ", math.log(math.pi)
```

When we run above program, it produces following result:

```
math.log(100.12) :  4.60636946656
math.log(100.72) :  4.61234438974
math.log(119L) :  4.77912349311
math.log(math.pi) :  1.14472988585
```

Example8: show the usage of log10() method by python codes.

```
import math   # This will import math module

print "math.log10(100.12) : ", math.log10(100.12)
print "math.log10(100.72) : ", math.log10(100.72)
print "math.log10(119L) : ", math.log10(119L)
print "math.log10(math.pi) : ", math.log10(math.pi)
```

When we run above program, it produces following result:

```
math.log10(100.12) :  2.00052084094
math.log10(100.72) :  2.0031157171
math.log10(119L) :  2.07554696139
math.log10(math.pi) :  0.497149872694
```

Example9: show the usage of max() method by python codes.

```
print "max(80, 100, 1000) : ", max(80, 100, 1000)
print "max(-20, 100, 400) : ", max(-20, 100, 400)
print "max(-80, -20, -10) : ", max(-80, -20, -10)
print "max(0, 100, -400) : ", max(0, 100, -400)
```

When we run above program, it produces following result:

```
max(80, 100, 1000) :  1000
max(-20, 100, 400) :  400
max(-80, -20, -10) :  -10
max(0, 100, -400) :  100
```

Example10: show the usage of min() method by python codes.

```
print "min(80, 100, 1000) : ", min(80, 100, 1000)
print "min(-20, 100, 400) : ", min(-20, 100, 400)
print "min(-80, -20, -10) : ", min(-80, -20, -10)
print "min(0, 100, -400) : ", min(0, 100, -400)
```

When we run above program, it produces following result:

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

Example11: show the usage of modf() method by python codes.

```
import math   # This will import math module

print "math.modf(100.12) : ", math.modf(100.12)
print "math.modf(100.72) : ", math.modf(100.72)
print "math.modf(119L) : ", math.modf(119L)
print "math.modf(math.pi) : ", math.modf(math.pi)
```

When we run above program, it produces following result:

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.71999999999999886, 100.0)
math.modf(119L) :    (0.0, 119.0)
```

```
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

Example12: show the usage of pow() method by python codes.

```
import math   # This will import math module

print "math.pow(100, 2) : ", math.pow(100, 2)
print "math.pow(100, -2) : ", math.pow(100, -2)
print "math.pow(2, 4) : ", math.pow(2, 4)
print "math.pow(3, 0) : ", math.pow(3, 0)
```

When we run above program, it produces following result:

```
math.pow(100, 2) :  10000.0
math.pow(100, -2) :  0.0001
math.pow(2, 4) :  16.0
math.pow(3, 0) :  1.0
```

Example13: show the usage of round() method by python codes.

```
print "round(80.23456, 2) : ", round(80.23456, 2)
print "round(100.000056, 3) : ", round(100.000056, 3)
print "round(-100.000056, 3) : ", round(-100.000056, 3)
```

When we run above program, it produces following result:

```
round(80.23456, 2) :  80.23
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

Example14: show the usage of sqrt() method by python codes.

```
import math   # This will import math module

print "math.sqrt(100) : ", math.sqrt(100)
print "math.sqrt(7) : ", math.sqrt(7)
print "math.sqrt(math.pi) : ", math.sqrt(math.pi)
```

When we run above program, it produces following result:

```
math.sqrt(100) :  10.0
math.sqrt(7) :  2.64575131106
math.sqrt(math.pi) :  1.77245385091
```

Example15: show the usage of choice() method by python codes.

```
import random
print "choice([1, 2, 3, 5, 9]) : ", random.choice([1, 2, 3, 5, 9])
print "choice('A String') : ", random.choice('A String')
```

When we run above program, it produces following result:

```
choice([1, 2, 3, 5, 9]) :  2
choice('A String') :  n
```

Example16: show the usage of randrange() method by python codes.

```
import random

# Select an even number in 100 <= number < 1000
print "randrange(100, 1000, 2) : ", random.randrange(100, 1000, 2)

# Select another number in 100 <= number < 1000
print "randrange(100, 1000, 3) : ", random.randrange(100, 1000, 3)
```

When we run above program, it produces following result:

```
randrange(100, 1000, 2) :  976
randrange(100, 1000, 3) :  520
```

Example17: show the usage of random () method by python codes.

```
import random

# First random number
print "random() : ", random.random()

# Second random number
print "random() : ", random.random()
```

When we run above program, it produces following result:

```
random() :  0.281954791393
random() :  0.309090465205
```

Example18: show the usage of seed () method by python codes.

```
import random

random.seed( 10 )
print "Random number with seed 10 : ", random.random()

# It will generate same random number
random.seed( 10 )
print "Random number with seed 10 : ", random.random()

# It will generate same random number
random.seed( 10 )
print "Random number with seed 10 : ", random.random()
```

When we run above program, it produces following result:

```
Random number with seed 10 :  0.57140259469
Random number with seed 10 :  0.57140259469
Random number with seed 10 :  0.57140259469
```

Example19: show the usage of shuffle () method by python codes.

```
import random

list = [20, 16, 10, 5];
random.shuffle(list)
print "Reshuffled list : ",  list
```

```
random.shuffle(list)
print "Reshuffled list : ",  list
```

When we run above program, it produces following result:

```
Reshuffled list :  [16, 5, 10, 20]
Reshuffled list :  [16, 5, 20, 10]
```

Example20: show the usage of uniform () method by python codes.

```
import random

print "Random Float uniform(5, 10) : ",  random.uniform(5, 10)

print "Random Float uniform(7, 14) : ",  random.uniform(7, 14)
```

Let us run the above program, this will produce the following result:

```
Random Float uniform(5, 10) :  5.52615217015
Random Float uniform(7, 14) :  12.5326369199
```

Example21: show the usage of acos() method by python codes.

```
import math

print "acos(0.64) : ",  math.acos(0.64)
print "acos(0) : ",  math.acos(0)
print "acos(-1) : ",  math.acos(-1)
print "acos(1) : ",  math.acos(1)
```

When we run above program, it produces following result:

```
acos(0.64) :  0.876298061168
acos(0) :  1.57079632679
acos(-1) :  3.14159265359
acos(1) :  0.0
```

Example22: show the usage of asin () method by python codes.

```
import math

print "asin(0.64) : ",  math.asin(0.64)
print "asin(0) : ",  math.asin(0)
print "asin(-1) : ",  math.asin(-1)
print "asin(1) : ",  math.asin(1)
```

When we run above program, it produces following result:

```
asin(0.64) :  0.694498265627
asin(0) :  0.0
asin(-1) :  -1.57079632679
asin(1) :  1.57079632679
```

Example23: show the usage of atan () method by python codes.

```
import math
```

```
print "atan(0.64) : ",  math.atan(0.64)
print "atan(0) : ",  math.atan(0)
print "atan(10) : ",  math.atan(10)
print "atan(-1) : ",  math.atan(-1)
print "atan(1) : ",  math.atan(1)
```

When we run above program, it produces following result:

```
atan(0.64) :  0.569313191101
atan(0) :  0.0
atan(10) :  1.4711276743
atan(-1) :  -0.785398163397
atan(1) :  0.785398163397
```

Example24: show the usage of atan2 () method by python codes.

```
import math

print "atan2(-0.50,-0.50) : ",  math.atan2(-0.50,-0.50)
print "atan2(0.50,0.50) : ",  math.atan2(0.50,0.50)
print "atan2(5,5) : ",  math.atan2(5,5)
print "atan2(-10,10) : ",  math.atan2(-10,10)
print "atan2(10,20) : ",  math.atan2(10,20)
```

When we run above program, it produces following result:

```
atan2(-0.50,-0.50) :  -2.35619449019
atan2(0.50,0.50) :  0.785398163397
atan2(5,5) :  0.785398163397
atan2(-10,10) :  -0.785398163397
atan2(10,20) :  0.463647609001
```

Example25: show the usage of cos () method by python codes.

```
import math

print "cos(3) : ",  math.cos(3)
print "cos(-3) : ",  math.cos(-3)
print "cos(0) : ",  math.cos(0)
print "cos(math.pi) : ",  math.cos(math.pi)
print "cos(2*math.pi) : ",  math.cos(2*math.pi)
```

When we run above program, it produces following result:

```
cos(3) :  -0.9899924966
cos(-3) :  -0.9899924966
cos(0) :  1.0
cos(math.pi) :  -1.0
cos(2*math.pi) :  1.0
```

Example26: show the usage of hypot () method by python codes.

```
import math

print "hypot(3, 2) : ",  math.hypot(3, 2)
```

```
print "hypot(-3, 3) : ",  math.hypot(-3, 3)
print "hypot(0, 2) : ",  math.hypot(0, 2)
```

When we run above program, it produces following result:

```
hypot(3, 2) :  3.60555127546
hypot(-3, 3) :  4.24264068712
hypot(0, 2) :  2.0
```

Example27: show the usage of sin () method by python codes.

```
#!/usr/bin/python
import math

print "sin(3) : ",  math.sin(3)
print "sin(-3) : ",  math.sin(-3)
print "sin(0) : ",  math.sin(0)
print "sin(math.pi) : ",  math.sin(math.pi)
print "sin(math.pi/2) : ",  math.sin(math.pi/2)
```

When we run above program, it produces following result:

```
sin(3) :  0.14112000806
sin(-3) :  -0.14112000806
sin(0) :  0.0
sin(math.pi) :  1.22464679915e-16
sin(math.pi/2) :  1.0
```

Example28: show the usage of tan () method by python codes.

```
import math

print "tan(3) : ",  math.tan(3)
print "tan(-3) : ",  math.tan(-3)
print "tan(0) : ",  math.tan(0)
print "tan(math.pi) : ",  math.tan(math.pi)
print "tan(math.pi/2) : ",  math.tan(math.pi/2)
print "tan(math.pi/4) : ",  math.tan(math.pi/4)
```

When we run above program, it produces following result:

```
tan(3) :  -0.142546543074
tan(-3) :  0.142546543074
tan(0) :  0.0
tan(math.pi) :  -1.22460635382e-16
tan(math.pi/2) :  1.63317787284e+16
tan(math.pi/4) :  1.0
```

Example29: show the usage of degrees () method by python codes.

```
import math

print "degrees(3) : ",  math.degrees(3)
print "degrees(-3) : ",  math.degrees(-3)
print "degrees(0) : ",  math.degrees(0)
print "degrees(math.pi) : ",  math.degrees(math.pi)
print "degrees(math.pi/2) : ",  math.degrees(math.pi/2)
print "degrees(math.pi/4) : ",  math.degrees(math.pi/4)
```

When we run above program, it produces following result:

```
degrees(3) :  171.887338539
degrees(-3) :  -171.887338539
degrees(0) :  0.0
degrees(math.pi) :  180.0
degrees(math.pi/2) :  90.0
degrees(math.pi/4) :  45.0
```

Example30: show the usage of radians () method by python codes.

```
import math

print "radians(3) : ",  math.radians(3)
print "radians(-3) : ",  math.radians(-3)
print "radians(0) : ",  math.radians(0)
print "radians(math.pi) : ",  math.radians(math.pi)
print "radians(math.pi/2) : ",  math.radians(math.pi/2)
print "radians(math.pi/4) : ",  math.radians(math.pi/4)
```

When we run above program, it produces following result:

```
radians(3) :  0.0523598775598
radians(-3) :  -0.0523598775598
radians(0) :  0.0
radians(math.pi) :  0.0548311355616
radians(math.pi/2) :  0.0274155677808
radians(math.pi/4) :  0.0137077838904
```

Exercise with solution: Write a Python program to guess a number between1 to 9.

Note : User is prompted to enter a guess. If the user guesses wrong then the prompt appears again until the guess is correct, on successful guess, user will get a "Well guessed!" message, and the program will exit.

Solution

```
import random
target_num, guess_num = random.randint(1, 10), 0
while target_num != guess_num:
    guess_num = int(input('Guess a number between 1 and 10 until you get it
right : '))
print('Well guessed!')
```

Output:

```
Guess a number between 1 and 10 until you get it right : 5       Well guessed!
```

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

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

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

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

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

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

```
Updated String :-  Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace

\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase r or uppercase R and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf family. Following is a simple example −

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

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

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer *lowercaseletters*
%X	hexadecimal integer *UPPERcaseletters*
%e	exponential notation *withlowercase'e'*
%E	exponential notation *withUPPERcase'E'*
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero *'0'* or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros *insteadofspaces*
%	'%%' leaves you with a single literal '%'

var	mapping variable *dictionaryarguments*
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point *ifappl.*

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code \n −

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (    ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it −

```
#!/usr/bin/python

print 'C:\\nowhere'
```

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

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows −

```
#!/usr/bin/python

print r'C:\\nowhere'
```

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

```
C:\\nowhere
```

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following −

```
#!/usr/bin/python

print u'Hello, world!'
```

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

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

Built-in String Methods

Python includes the following built-in methods to manipulate strings −

SN	Methods with Description
1	capitalize Capitalizes first letter of string
2	center*width, fillchar* Returns a space-padded string with the original string centered to a total of width columns.
3	count*str, beg = 0, end = len(string)* Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode*encoding = 'UTF − 8', errors = 'strict'* Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	encode*encoding = 'UTF − 8', errors = 'strict'* Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	endswith*suffix, beg = 0, end = len(string)* Determines if string or a substring of string *ifstartingindexbegandendingindexendaregiven* ends with suffix; returns true if so and false otherwise.
7	expandtabs*tabsize = 8* Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	find*str, beg = 0end = len(string)*

Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

9

index*str, beg = 0, end = len(string)*

Same as find, but raises an exception if str not found.

10

isalnum

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

11

isalpha

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

12

isdigit

Returns true if string contains only digits and false otherwise.

13

islower

Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

14

isnumeric

Returns true if a unicode string contains only numeric characters and false otherwise.

15

isspace

Returns true if string contains only whitespace characters and false otherwise.

16

istitle

Returns true if string is properly "titlecased" and false otherwise.

17

isupper

Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.

18

join*seq*

Merges *concatenates* the string representations of elements in sequence seq into a string, with separator string.

19

len*string*

Returns the length of the string

20

ljust*width*[, *fillchar*]

Returns a space-padded string with the original string left-justified to a total of width columns.

21

lower

Converts all uppercase letters in string to lowercase.

22

lstrip

Removes all leading whitespace in string.

23

maketrans

Returns a translation table to be used in translate function.

24

max*str*

Returns the max alphabetical character from the string str.

25

min*str*

Returns the min alphabetical character from the string str.

26

replace*old, new*[, *max*]

Replaces all occurrences of old in string with new or at most max occurrences if max given.

27

rfind*str, beg = 0, end = len(string)*

Same as find, but search backwards in string.

28

rindex*str, beg = 0, end = len(string)*

Same as index, but search backwards in string.

29

rjust*width,* [, *fillchar*]

Returns a space-padded string with the original string right-justified to a total of width columns.

30

rstrip

Removes all trailing whitespace of string.

31

splitstr="", num=string.count(str)

Splits string according to delimiter str space if not provided and returns list of substrings; split into at most num substrings if given.

32

splitlines num=string.count('\n')

Splits string at all or num NEWLINEs and returns a list of each line with NEWLINEs removed.

33

startswithstr, beg=0,end=len(string)

Determines if string or a substring of string if starting index beg and ending index end are given starts with substring str; returns true if so and false otherwise.

34

strip[chars]

Performs both lstrip and rstrip on string

35

swapcase

Inverts case for all letters in string.

36

title

Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

37

translatetable, deletechars=""

Translates string according to translation table str256 chars, removing those in the del string.

38

upper

Converts lowercase letters in string to uppercase.

39

zfill width

Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill retains any sign given less one zero.

40

isdecimal

Returns true if a unicode string contains only decimal characters and false otherwise.

Examples Sheet #4 (Python Strings)

Example1: show the usage of capitalize() method by python codes.

```
str = "this is string example....wow!!!";
print "str.capitalize() : ", str.capitalize()
```

Result

```
str.capitalize() :  This is string example....wow!!!
```

Example2: show the usage of center() method by python codes.

```
str = "this is string example....wow!!!";
print "str.center(40, 'a') : ", str.center(40, 'a')
```

Result

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

Example3: show the usage of count() method by python codes.

```
str = "this is string example....wow!!!";
sub = "i";
print "str.count(sub, 4, 40) : ", str.count(sub, 4, 40)
sub = "wow";
print "str.count(sub) : ", str.count(sub)
```

Result

```
str.count(sub, 4, 40) :  2
str.count(sub) :  1
```

Example4: show the usage of decode() method by python codes.

```
Str = "this is string example....wow!!!";
Str = Str.encode('base64','strict');
print "Encoded String: " + Str
print "Decoded String: " + Str.decode('base64','strict')
```

Result

```
Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=

Decoded String: this is string example....wow!!!
```

Example5: show the usage of encode() method by python codes.

```
str = "this is string example....wow!!!";
print "Encoded String: " + str.encode('base64','strict')
```

Result

```
Encoded String: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=
```

Example6: show the usage of endswith() method by python codes.

```
str = "this is string example....wow!!!";
suffix = "wow!!!";
print str.endswith(suffix)
print str.endswith(suffix,20)
```

```
suffix = "is";
print str.endswith(suffix, 2, 4)
print str.endswith(suffix, 2, 6)
```

Result

```
True
True
True
False
```

Example7: show the usage of expandtabs() method by python codes.

```
str = "this is\tstring example....wow!!!";
print "Original string: " + str
print "Defualt exapanded tab: " +  str.expandtabs()
print "Double exapanded tab: " +  str.expandtabs(16)
```

Result

```
Original string: this is	string example....wow!!!
Defualt exapanded tab: this is string example....wow!!!
Double exapanded tab: this is         string example....wow!!!
```

Example8: show the usage of find() method by python codes.

```
str1 = "this is string example....wow!!!";
str2 = "exam";
print str1.find(str2)
print str1.find(str2, 10)
print str1.find(str2, 40)
```

Result

```
15
15
-1
```

Example9: show the usage of index () method by python codes.

```
str1 = "this is string example....wow!!!";
str2 = "exam";
print str1.index(str2)
print str1.index(str2, 10)
print str1.index(str2, 40) # error, not fount after this starting index
```

Result

```
15
15
Traceback (most recent call last):
  File "test.py", line 8, in
  print str1.index(str2, 40);
ValueError: substring not found
shell returned 1
```

Example10: show the usage of isalnum() method by python codes.

```
str = "this2009";  # No space in this string
print str.isalnum()
str = "this is string example....wow!!!";
print str.isalnum()
```

When we run above program, it produces following result:

```
True
False
```

Example11: show the usage of isalpha() method by python codes.

```
str = "this";  # No space & digit in this string
print str.isalpha()
str = "this is string example....wow!!!";
print str.isalpha()
```

When we run above program, it produces following result –

```
True
False
```

Example12: show the usage of isdigit() method by python codes.

```
str = "123456";  # Only digit in this string
print str.isdigit()
str = "this is string example....wow!!!";
print str.isdigit()
```

When we run above program, it produces following result –

```
True
False
```

Example13: show the usage of islower() method by python codes.

```
str = "THIS is string example....wow!!!";
print str.islower()
str = "this is string example....wow!!!";
print str.islower()
```

When we run above program, it produces following result –

```
False
True
```

Example14: show the usage of isnumeric() method by python codes.

```
str = u"this2009";
print str.isnumeric()
str = u"23443434";
print str.isnumeric()
```

When we run above program, it produces following result –

```
False
True
```

Example15: show the usage of isspace() method by python codes.

```
str = "       ";
print str.isspace()
str = "This is string example....wow!!!";
print str.isspace()
```

When we run above program, it produces following result –

```
True
False
```

Example16: show the usage of istitle() method by python codes.

```
str = "This Is String Example...Wow!!!";
print str.istitle()
str = "This is string example....wow!!!";
print str.istitle()
```

When we run above program, it produces following result –

```
True
False
```

Example17: show the usage of isupper() method by python codes.

```
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.isupper()
str = "THIS is string example....wow!!!";
print str.isupper()
```

When we run above program, it produces following result –

```
True
False
```

Example18: show the usage of join() method by python codes.

```
s = "-";
seq = ("a", "b", "c"); # This is sequence of strings.
print s.join( seq )
```

When we run above program, it produces following result –

```
a-b-c
```

Example19: show the usage of len() method by python codes.

```
str = "this is string example....wow!!!";
print "Length of the string: ", len(str)
```

When we run above program, it produces following result –

```
Length of the string:  32
```

Example20: show the usage of ljust() method by python codes.

```
str = "this is string example....wow!!!";
```

```
print str.ljust(50, '0')
```

When we run above program, it produces following result –

```
this is string example....wow!!!000000000000000000
```

Example21: show the usage of lower() method by python codes.

```
str = "THIS IS STRING EXAMPLE....WOW!!!";

print str.lower()
```

When we run above program, it produces following result –

```
this is string example....wow!!!
```

Example22: show the usage of lstrip() method by python codes.

```
str = "     this is string example....wow!!!     ";
print str.lstrip()
str = "88888888this is string example....wow!!!8888888";
print str.lstrip('8')
```

When we run above program, it produces following result –

```
this is string example....wow!!!
this is string example....wow!!!8888888
```

Example23: show the usage of maketrans() method by python codes.

```
from string import maketrans   # Required to call maketrans function.
intab = "aeiou"
outtab = "12345"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!"
print str.translate(trantab)
```

When we run above program, it produces following result –

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

Example24: show the usage of max() method by python codes.

```
str = "this is really a string example....wow!!!";
print "Max character: " + max(str)
str = "this is a string example....wow!!!";
print "Max character: " + max(str)
```

When we run above program, it produces following result –

```
Max character: y
Max character: x
```

Example25: show the usage of min() method by python codes.

```
str = "this-is-real-string-example....wow!!!";
print "Min character: " + min(str)
str = "this-is-a-string-example....wow!!!";
print "Min character: " + min(str)
```

When we run above program, it produces following result –

```
Min character: !
Min character: !
```

Example26: show the usage of replace() method by python codes.

```
str = "this is string example....wow!!! this is really string"
print str.replace("is", "was")
print str.replace("is", "was", 3)
```

When we run above program, it produces following result –

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

Example27: show the usage of rfind() method by python codes.

```
str1 = "this is really a string example....wow!!!";
str2 = "is";
print str1.rfind(str2)
print str1.rfind(str2, 0, 10)
print str1.rfind(str2, 10, 0)
print str1.find(str2)
print str1.find(str2, 0, 10)
print str1.find(str2, 10, 0)
```

When we run above program, it produces following result –

```
5
5
-1
2
2
-1
```

Example28: show the usage of rindex() method by python codes.

```
str1 = "this is string example....wow!!!";
str2 = "is";
print str1.rindex(str2)
print str1.index(str2)
```

When we run above program, it produces following result –

```
5
2
```

Example29: show the usage of rjust() method by python codes.

```
str = "this is string example....wow!!!";
print str.rjust(50, '0')
```

When we run above program, it produces following result –

```
000000000000000000this is string example....wow!!!
```

Example30: show the usage of rstrip() method by python codes.

```
str = "     this is string example....wow!!!     ";
print str.rstrip()
```

```
str = "88888888this is string example....wow!!!8888888";
print str.rstrip('8')
```

When we run above program, it produces following result –

```
     this is string example....wow!!!
88888888this is string example....wow!!!
```

Example31: show the usage of split() method by python codes.

```
str = "Line1-abcdef \nLine2-abc \nLine4-abcd";
print str.split( )
print str.split(' ', 1 )
```

When we run above program, it produces following result –

```
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
```

Example32: show the usage of splitlines() method by python codes.

```
str = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d";
print str.splitlines( )
print str.splitlines( 0 )
print str.splitlines( 3 )
print str.splitlines( 4 )
print str.splitlines( 5 )
```

When we run above program, it produces following result –

```
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
```

Example33: show the usage of startswith() method by python codes.

```
str = "this is string example....wow!!!";
print str.startswith( 'this' )
print str.startswith( 'is', 2, 4 )
print str.startswith( 'this', 2, 4 )
```

When we run above program, it produces following result –

```
True
True
False
```

Example34: show the usage of strip () method by python codes.

```
str = "0000000this is string example....wow!!!0000000";
print str.strip( '0' )
```

When we run above program, it produces following result –

```
this is string example....wow!!!
```

Example35: show the usage of swapcase() method by python codes.

```
str = "this is string example....wow!!!";
```

```
print str.swapcase()
str = "THIS IS STRING EXAMPLE....WOW!!!";
print str.swapcase()
```

When we run above program, it produces following result –

```
THIS IS STRING EXAMPLE....WOW!!!
this is string example....wow!!!
```

Example36: show the usage of title() method by python codes.

```
str = "this is string example....wow!!!";
print str.title()
```

When we run above program, it produces following result –

```
This Is String Example....Wow!!!
```

Example37: show the usage of translate() method by python codes.

```
from string import maketrans   # Required to call maketrans function.
intab = "aeiou"
outtab = "12345"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print str.translate(trantab)
```

When we run above program, it produces following result –

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

Following is the example to delete 'x' and 'm' characters from the string –

```
from string import maketrans   # Required to call maketrans function.
intab = "aeiou"
outtab = "12345"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print str.translate(trantab, 'xm')
```

This will produce following result –

```
th3s 3s str3ng 21pl2....w4w!!!
```

Example38: show the usage of upper() method by python codes.

```
str = "this is string example....wow!!!";

print "str.capitalize() : ", str.upper()
```

When we run above program, it produces following result –

```
str.capitalize() :  THIS IS STRING EXAMPLE....WOW!!!
```

Example39: show the usage of zfill() method by python codes.

```
str = "this is string example....wow!!!";

print str.zfill(40)
```

```
print str.zfill(50)
```

When we run above program, it produces following result –

```
00000000this is string example....wow!!!
0000000000000000000this is string example....wow!!!
```

Example40: show the usage of isdecimal() method by python codes.

```
str = u"this2009";
print str.isdecimal();
str = u"23443434";
print str.isdecimal();
```

When we run above program, it produces following result –

```
False
True
```

Exercises with Solutions

1-Write a Python program to check whether a string starts with specified characters.

```
#Solution
string = "w3resource.com"
print(string.startswith("w3r"))
#Output:True
```

2-Write a Python program to remove a newline in Python.

```
#Solution
str1='Python Exercises\n'
print(str1)
print(str1.rstrip())
#Output:
"""
Python Exercises

Python Exercises
"""
```

3-Write a Python function to reverses a string if it's length is a multiple of 4

```
#Solution
str1='abcd'
#str1='python'
if len(str1) % 4 == 0:
   print ''.join(reversed(str1))
print str1
#Output: dcba
#Output: python
```

4-Write a Python program to remove the characters which have odd index values of a given string.

```
#Solution
str='abcdef'
#str='python'
result = ""
for i in range(len(str)):
  if i % 2 == 0:
    result = result + str[i]
print result
#Output: ace
#Output: pto
```

5-Write a Python script that takes input from the user and displays that input back in upper and lower cases.

```
#Solution
user_input = input("What's your favourite language? ")
print("My favourite language is ", user_input.upper())
print("My favourite language is ", user_input.lower())
#Output:
"""
What's your favourite language? english
My favourite language is  ENGLISH
My favourite language is  english
"""
```

6-Write a Python program to get substring before last specified character.

```
#Solution
str1 = 'http://www.w3resource.com/python-exercises/string'
print(str1.rsplit('/', 1)[0])#The substring before last character '/'
print(str1.rsplit('-', 1)[0])#The substring before last character '_'
#Output:
#http://www.w3resource.com/python-exercises
#http://www.w3resource.com/python
```

H.W: What is the output of the following code?

Choose one: (PYTHON, PYTHONSTRING, ' ', STRING)

```
ch=' '
for char in 'PYTHON STRING':
        ch=ch+char
        if char == ' ':
                print ch
                break
```

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

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

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

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

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

Accessing Values in Lists

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

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

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

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

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

```
#!/usr/bin/python

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

print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Note: append method is discussed in subsequent section.

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

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which elements you are deleting or the remove method if you do not know. For example −

```
#!/usr/bin/python

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

print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

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

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Note: remove method is discussed in subsequent section.

Basic List Operations

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

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

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

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

```
L = ['spam', 'Spam', 'SPAM!']
```

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

Built-in List Functions & Methods:

Python includes the following list functions −

SN	Function with Description
1	cmp*list1, list2* Compares elements of both lists.
2	len*list* Gives the total length of the list.
3	max*list* Returns item from the list with max value.
4	min*list* Returns item from the list with min value.
5	list*seq* Converts a tuple into list.

Python includes following list methods

SN	Methods with Description
1	list.append*obj* Appends object obj to list
2	list.count*obj* Returns count of how many times obj occurs in list
3	list.extend*seq*

	Appends the contents of seq to list	
4	list.index*obj*	
	Returns the lowest index in list that obj appears	
5	list.insert*index, obj*	
	Inserts object obj into list at offset index	
6	list.pop*obj = list*[− 1]	
	Removes and returns last object or obj from list	
7	list.remove*obj*	
	Removes object obj from list	
8	list.reverse	
	Reverses objects of list in place	
9	list.sort[*func*]	
	Sorts objects of list, use compare func if given	

Examples Sheet #5 (Python Lists)

Example1: show the usage of cmp() method by python codes.

```
list1, list2 = [123, 'xyz'], [456, 'abc']
print cmp(list1, list2)
print cmp(list2, list1)
list3 = list2 + [786];
print cmp(list2, list3)
```

When we run above program, it produces following result –

```
-1
1
-1
```

Example2: show the usage of len() method by python codes.

```
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print "First list length : ", len(list1)
print "Second list length : ", len(list2)
```

When we run above program, it produces following result –

```
First list length :  3
Second list length :  2
```

Example3: show the usage of max() method by python codes.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "Max value element : ", max(list1)
print "Max value element : ", max(list2)
```

When we run above program, it produces following result –

```
Max value element :  zara
Max value element :  700
```

Example4: show the usage of min() method by python codes.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
print "min value element : ", min(list1)
print "min value element : ", min(list2)
```

When we run above program, it produces following result –

```
min value element :  123
min value element :  200
```

Example5: show the usage of list() method by python codes.

```
aTuple = (123, 'xyz', 'zara', 'abc');
aList = list(aTuple)
print "List elements : ", aList
```

When we run above program, it produces following result:

```
List elements :  [123, 'xyz', 'zara', 'abc']
```

Example6: show the usage of append() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc'];
aList.append( 2009 );
print "Updated List : ", aList
```

When we run above program, it produces following result –

```
Updated List :  [123, 'xyz', 'zara', 'abc', 2009]
```

Example7: show the usage of count() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc', 123];
print "Count for 123 : ", aList.count(123)
print "Count for zara : ", aList.count('zara')
```

When we run above program, it produces following result –

```
Count for 123 :  2
Count for zara :  1
```

Example8: show the usage of extend() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc', 123];
bList = [2009, 'manni'];
aList.extend(bList)
print "Extended List : ", aList
```

When we run above program, it produces following result –

```
Extended List :  [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

Example9: show the usage of index() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc'];
print "Index for xyz : ", aList.index( 'xyz' )
print "Index for zara : ", aList.index( 'zara' )
```

When we run above program, it produces following result –

```
Index for xyz :  1
Index for zara :  2
```

Example10: show the usage of insert() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc']
aList.insert( 3, 2009)
print "Final List : ", aList
```

When we run above program, it produces following result –

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

Example11: show the usage of pop() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc'];
print "A List : ", aList.pop()
print "B List : ", aList.pop(2)
```

When we run above program, it produces following result –

```
A List :  abc
B List :  zara
```

Example12: show the usage of remove() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print "List : ", aList
aList.remove('abc');
print "List : ", aList
```

When we run above program, it produces following result –

```
List :  [123, 'zara', 'abc', 'xyz']
List :  [123, 'zara', 'xyz']
```

Example13: show the usage of reverse() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print "List : ", aList
```

When we run above program, it produces following result –

```
List :  ['xyz', 'abc', 'zara', 'xyz', 123]
```

Example14: show the usage of sort() method by python codes.

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.sort();
print "List : ", aList
```

When we run above program, it produces following result –

```
List :  [123, 'abc', 'xyz', 'xyz', 'zara']
```

Exercises with Solutions

1-Write a Python program to sum all the items in a list=[1,2,-8].

```
#Solution1:
Print sum([1,2,-8])
#Output: -5

#Solution2:
items=[1,2,-8]
sum_numbers = 0
for x in items:
    sum_numbers += x
print "sum_numbers=", sum_numbers
#Output: sum_numbers=-5
```

2-Write a Python program to get the largest number from a list=[1, 2, -8, 0].

```
#Solution1:
```

```
Print max([1, 2, -8, 0])
#Output: 2
#Solution2:
max = list[0]
for a in list:
    if a > max:
        max = a
print max
#Output: 2
```

3-Write a Python program to remove duplicates from a list=[10,20,30,20,10,50,60,40,80,50,40].

```
#Solution:
a=[10,20,30,20,10,50,60,40,80,50,40]
b=[]
for i in a:
    if i not in b:
        b.append(i)
print b
#Output: [10, 20, 30, 50, 60, 40, 80]
#H.W: How to get only the duplicated list=[20, 10, 50, 40]
```

4-Write a Python program to check a list is empty or not.

```
#Solution:
l = []
if not l:
  print "List is empty"
#Output: "List is empty"
```

5- Write a Python program to select an item randomly from a list= `['Red', 'Blue', 'Green', 'White', 'Black']`

```
#Solution:
import random
color_list = ['Red', 'Blue', 'Green', 'White', 'Black']
print(random.choice(color_list))
#output: Black
```

6- Write a Python program to find common items from two lists:

```
#Solution:
list1=["Red", "Green", "Orange", "White"]
list2=["Black", "Green", "White", "Pink"]
print list(set(list1)&set(list2))
#Output: ['Green', 'White']
```

7-Write a Python program to perform the following on the list,x= [10, 20, 30, 40, 50, 60, 70, 80, 90].

1- Get first two elements.

```
#Solution:print(x[:2])#Output:[10, 20]
```

2- Get last two elements.

```
#Solution:print(x[-2:])#Output:[80, 90]
```

3- Get elements after the first two elements.

```
#Solution:print(x[2:])#Output:[30, 40, 50, 60, 70, 80, 90]
```

4- Get elements before the last two elements.

```
#Solution:print(x[:-2])#Output:[10, 20, 30, 40, 50, 60, 70]
```

5- Get the elements in the odd locations of a list.

```
#Solution:print(x[::2]) # Output:[10, 30, 50, 70, 90]
```

8- Write a Python program to find the second smallest number in a list=[1, 2, -8, -2, 0]

```
#Solution:
 a1, a2 = float('inf'), float('inf')
 for x in [1, 2, -8, -2, 0]:
     if x <= a1:
         a1, a2 = x, a1
     elif x < a2:
         a2 = x
print a2
#output: -2
```

9-Write a Python program to generate all sublists of a list.

```
#Solution:
my_list=[10, 20, 30, 40]
#my_list=['X', 'Y', 'Z']
subs = [[]]
for i in range(len(my_list)):
        n = i+1
        while n <= len(my_list):
                sub = my_list[i:n]
                subs.append(sub)
                n += 1
print subs
#output:[[], [10], [10, 20], [10, 20, 30], [10, 20, 30, 40], [20], [20, 30],
[20, 30, 40], [30], [30, 40], [40]]
#output:[[], ['X'], ['X', 'Y'], ['X', 'Y', 'Z'], ['Y'], ['Y', 'Z'], ['Z']]
```

10- Write a Python program to check whether a list contains a sublist, for example does `[4,3]` in `[2,4,3,5,7]`? and does `[3,7]` in `[2,4,3,5,7]`?

```
#Solution:
l=[2,4,3,5,7]
s=[4,3]
#s=[3,7]
sub_set = False
if s == []:
        sub_set = True
elif s == l:
        sub_set = True
elif len(s) > len(l):
        sub_set = False
else:
        for i in range(len(l)):
                if l[i] == s[0]:
                        n = 1
                        while (n < len(s)) and (l[i+n] == s[n]):
```

```
                                n += 1
                        if n == len(s):
                                sub_set = True
print sub_set
#output: True for s=[4,3] and false for s=[3,7]
```

H.W1: Write a Python program to extend a list without append.
Input lists: [10, 20, 30], [40, 50, 60]
New List: [40, 50, 60, 10, 20, 30]
H.W2: Write a Python program to remove duplicates from a list of lists.
Input list : [[10, 20], [40], [30, 56, 25], [10, 20], [33], [40]]
New List : [[10, 20], [30, 56, 25], [33], [40]]
H.W3: Write a Python program to find the list in a list of lists whose sum of elements is the highest.
Input lists: [1,2,3], [4,5,6], [10,11,12], [7,8,9]
Output: [10, 11, 12]

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

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

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

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

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

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

Accessing Values in Tuples:

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

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

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

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples

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

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3
```

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

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

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example:

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup
del tup;
print "After deleting tup : "
print tup
```

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

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

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

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

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

Indexing, Slicing, and Matrixes

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

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

Python Expression	Results	Description

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

No Enclosing Delimiters

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

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz'
x, y = 1, 2;
print "Value of x , y : ", x,y
```

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

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions −

SN	Function with Description
1	cmp*tuple1, tuple2* Compares elements of both tuples.
2	len*tuple* Gives the total length of the tuple.
3	max*tuple* Returns item from the tuple with max value.
4	min*tuple* Returns item from the tuple with min value.
5	tuple*seq* Converts a list into tuple.

Examples Sheet #6 (Python Tuples)

Example1: show the usage of cmp() method by python codes.

```
tuple1, tuple2 = (123, 'xyz'), (456, 'abc')
print cmp(tuple1, tuple2)
print cmp(tuple2, tuple1)
tuple3 = tuple2 + (786,);
print cmp(tuple2, tuple3)
```

When we run above program, it produces following result –

```
-1
1
-1
```

Example2: show the usage of len() method by python codes.

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print "First tuple length : ", len(tuple1)
print "Second tuple length : ", len(tuple2)
```

When we run above program, it produces following result –

```
First tuple length :  3
Second tuple length :  2
```

Example3: show the usage of max() method by python codes.

```
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
print "Max value element : ", max(tuple1)
print "Max value element : ", max(tuple2)
```

When we run above program, it produces following result –

```
Max value element :  zara
Max value element :  700
```

Example4: show the usage of min() method by python codes.

```
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
print "min value element : ", min(tuple1)
print "min value element : ", min(tuple2)
```

When we run above program, it produces following result –

```
min value element :  123
min value element :  200
```

Example5: show the usage of tuple () method by python codes.

```
aList = (123, 'xyz', 'zara', 'abc');
aTuple = tuple(aList)
print "Tuple elements : ", aTuple
```

When we run above program, it produces following result –

```
Tuple elements :  (123, 'xyz', 'zara', 'abc')
```

Exercises with Solutions

1. Write a Python program to create a tuple.

```
a) Create an empty tuple:
x = ()
print x #()
b) Create an empty tuple with tuple() function built-in Python:
tuplex = tuple()
print tuplex #()
c) Create a tuple with different data types:
tuplex = ("tuple", False, 3.2, 1)
print "tuplex=", tuplex # tuplex = ("tuple", False, 3.2, 1)
```

2. Write a Python program to add an item in a tuple.

```
Note: tuples are immutable, so you cannot add new elements directly.
a) create a tuple:
tuplex = (4, 6, 2, 8, 3, 1)
print tuplex #(4, 6, 2, 8, 3, 1)
b) using merge of tuples with the + operator you can add an element and it
will create a new tuple:
tuplex = tuplex + (9,)
print tuplex #(4, 6, 2, 8, 3, 1, 9)
c) adding items in a specific index:
tuplex = tuplex[:5] + (15, 20, 25) + tuplex[:5]
print tuplex # (4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3)
d) converting the tuple to list:
listx = list(tuplex) #[4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3]
e) use different ways to add items in list:
listx.append(30)
tuplex = tuple(listx)
print tuplex # (4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3, 30)
```

3- Write a Python program to convert a tuple to a string.

```
tup = ('e', 'x', 'e', 'r', 'c', 'i', 's', 'e', 's')
str =  ''.join(tup)
print str # exercises
```

H.W: Write a Python program to reverse a tuple.

```
Example:
Input: (5, 10, 15, 20)
Output:(20, 15, 10, 5)
```

Each key is separated from its value by a colon :, the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary:

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

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

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

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

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

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

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

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

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

```
dict['Zara']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

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

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry


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

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

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

Delete Dictionary Elements

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

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

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary

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

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note: del method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

a More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

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

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

```
dict['Name']:  Manni
```

b Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example:

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7};

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

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

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

Built-in Dictionary Functions & Methods −

Python includes the following dictionary functions −

SN	Function with Description
1	cmp*dict1, dict2* Compares elements of both dict.
2	len*dict* Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str*dict* Produces a printable string representation of a dictionary
4	type*variable* Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods −

SN	Methods with Description
1	dict.clear Removes all elements of dictionary *dict*
2	dict.copy Returns a shallow copy of dictionary *dict*
3	dict.fromkeys Create a new dictionary with keys from seq and values *set* to *value*.
4	dict.get*key, default = None* For *key* key, returns value or default if key not in dictionary
5	

	dict.has_key*key* Returns *true* if key in dictionary *dict*, *false* otherwise
6	dict.items Returns a list of *dict*'s *key*, *value* tuple pairs
7	dict.keys Returns list of dictionary dict's keys
8	dict.setdefault*key, default = None* Similar to get, but will set dict[key]=default if *key* is not already in dict
9	dict.update*dict2* Adds dictionary *dict2*'s key-values pairs to *dict*
10	dict.values Returns list of dictionary *dict*'s values

Examples Sheet #7 (Python Dictionary)

Example1: show the usage of cmp() method by python codes.

```
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = {'Name': 'Mahnaz', 'Age': 27};
dict3 = {'Name': 'Abid', 'Age': 27};
dict4 = {'Name': 'Zara', 'Age': 7};
print "Return Value : %d" %  cmp (dict1, dict2)
print "Return Value : %d" %  cmp (dict2, dict3)
print "Return Value : %d" %  cmp (dict1, dict4)
```

When we run above program, it produces following result –

```
Return Value : -1
Return Value : 1
Return Value : 0
```

Example2: show the usage of len() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7};
print "Length : %d" % len (dict)
```

When we run above program, it produces following result –

```
Length : 2
```

Example3: show the usage of str() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7};
print "Equivalent String : %s" % str (dict)
```

When we run above program, it produces following result –

```
Equivalent String : {'Age': 7, 'Name': 'Zara'}
```

Example4: show the usage of type() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7};
print "Variable Type : %s" %  type (dict)
```

When we run above program, it produces following result –

```
Variable Type : <type 'dict'>
```

Example5: show the usage of clear() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7};
print "Start Len : %d" %  len(dict)
dict.clear()
print "End Len : %d" %  len(dict)
```

When we run above program, it produces following result –

```
Start Len : 2
End Len : 0
```

Example6: show the usage of copy() method by python codes.

```
dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = dict1.copy()
print "New Dictinary : %s" %  str(dict2)
```

When we run above program, it produces following result –

```
New Dictinary : {'Age': 7, 'Name': 'Zara'}
```

Example7: show the usage of fromkeys() method by python codes.

```
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print "New Dictionary : %s" %  str(dict)
dict = dict.fromkeys(seq, 10)
print "New Dictionary : %s" %  str(dict)
```

When we run above program, it produces following result –

```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

Example8: show the usage of get() method by python codes.

```
dict = {'Name': 'Zabra', 'Age': 7}
print "Value : %s" %  dict.get('Age')
print "Value : %s" %  dict.get('Education', "Never")
```

When we run above program, it produces following result –

```
Value : 7
Value : Never
```

Example9: show the usage of has_key () method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %  dict.has_key('Age')
print "Value : %s" %  dict.has_key('Sex')
```

When we run above program, it produces following result –

```
Value : True
Value : False
```

Example10: show the usage of items() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %  dict.items()
```

When we run above program, it produces following result –

```
Value : [('Age', 7), ('Name', 'Zara')]
```

Example11: show the usage of keys() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" %  dict.keys()
```

When we run above program, it produces following result –

```
Value : ['Age', 'Name']
```

Example12: show the usage of setdefault () method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %  dict.setdefault('Age', None)
print "Value : %s" %  dict.setdefault('Sex', None)
```

When we run above program, it produces following result –

```
Value : 7
Value : None
```

Example13: show the usage of update() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print "Value : %s" %  dict
```

When we run above program, it produces following result –

```
Value : {'Age': 7, 'Name': 'Zara', 'Sex': 'female'}
```

Example14: show the usage of values() method by python codes.

```
dict = {'Name': 'Zara', 'Age': 7}
print "Value : %s" %  dict.values()
```

When we run above program, it produces following result –

```
Value : [7, 'Zara']
```

Exercises with Solutions

1- Write a Python script to add a key 2 and value 30 to a dictionary= {0:10, 1:20}

```
#Solution: d={0:10, 1:20}; d.update({2:30}); print d
#output: {0: 10, 1: 20, 2: 30}
```

2- Write a Python program to:
a) Sort a dictionary by key:

```
#Solution:
color_dict = {'red':'#FF0000','green':'#008000','black':'#000000',
              'white':'#FFFFFF'}
for key in sorted(color_dict):
    print"%s: %s" % (key, color_dict[key])
#output:
#black: #000000
#green: #008000
#red: #FF0000
#white: #FFFFFF
```

b) Sort (ascending and descending) a dictionary by value.

```
#Solution
```

```
import operator
d = {1: 2, 3: 4, 4: 3, 2: 1, 0: 0}
print('Original dictionary : ',d)
sorted_d = sorted(d.items(), key=operator.itemgetter(0))
print('Dictionary in ascending order by value : ',sorted_d)
sorted_d = sorted(d.items(), key=operator.itemgetter(0),reverse=True)
print('Dictionary in descending order by value : ',sorted_d)
#Output:
#Original dictionary :  {0: 0, 1: 2, 2: 1, 3: 4, 4: 3}
#Dictionary in ascending order by value : [(0, 0), (1, 2), (2, 1), (3, 4), (4,
3)]
#Dictionary in descending order by value: [(4, 3), (3, 4), (2, 1), (1, 2), (0,
0)]
```

3-Write a Python script to concatenate following dictionaries to create a new one. For example, let you have the following three dictionaries:
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
The result is one dictionary : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}.

```
#Solution
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
dic4 = {}
for d in (dic1, dic2, dic3): dic4.update(d)
print dic4
#Output: {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

4- Write a Python script to check if a given key already exists in a dictionary.

```
#Solution
d = {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
x=5
#x=9
if x in d:
    print 'Key is present in the dictionary'
else:
    print 'Key is not present in the dictionary'
#Output: Key is present in the dictionary
#Output: Key is not present in the dictionary
```

5- Write a Python script to merge two Python dictionaries.

```
#Solution
d1 = {'a': 100, 'b': 200}
d2 = {'x': 300, 'y': 200}
d = d1.copy()
d.update(d2)
print d
#Output: {'x': 300, 'y': 200, 'a': 100, 'b': 200}
```

6-Write a Python program to sum all the items in a dictionary.

```
#Solution
```

```
my_dict = {'data1':100,'data2':-54,'data3':247}
print(sum(my_dict.values()))
# Output: 293
```

7-Write a Python program to map two lists into a dictionary.

```
#Solution
keys = ['red', 'green', 'blue']
values = ['#FF0000','#008000', '#0000FF']
color_dictionary = dict(zip(keys, values))
print(color_dictionary)
# Output: {'green': '#008000', 'blue': '#0000FF', 'red': '#FF0000'}
```

8-Write a Python program to get the maximum and minimum value in a dictionary

```
#Solution
my_dict = {'x':500, 'y':5874, 'z': 560}
key_max = max(my_dict.keys(), key=(lambda k: my_dict[k]))
key_min = min(my_dict.keys(), key=(lambda k: my_dict[k]))
print 'Maximum Value: ',my_dict[key_max]
print 'Minimum Value: ',my_dict[key_min]
# Output: Maximum Value:  5874
# Output: Minimum Value:  500
```

9-Write a Python program to remove duplicates from Dictionary.

```
#Solution
student_data = {'id1': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['Sara']},
                'id2': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['David']},
                'id3': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['Sara']},
                'id4': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['Surya']}}
result = {}
for key,value in student_data.items():
    if value not in result.values():
        result[key] = value
print " student_data =",result
#Output:
"""
student_data = {'id1': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['Sara']},
                'id2': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
                                        'name': ['David']},
                'id4': {'subject_integration': ['Python', 'Perl', 'Prolog'],
                                       'class': ['V'],
```

```
                                    'name': ['Surya']}}
"""
```

10-Write a Python program to create a dictionary from a string.
Note: Track the count of the letters from the string.
Sample string : 'w3resource'
output: {'3': 1, 's': 1, 'r': 2, 'u': 1, 'w': 1, 'c': 1, 'e': 2, 'o': 1}.

```
#Solution
from collections import defaultdict, Counter
str1 = 'w3resource'
my_dict = {}
for letter in str1:
    my_dict[letter] = my_dict.get(letter, 0) + 1
print(my_dict)
#Output:{'o': 1, '3': 1, 's': 1, 'r': 2, 'w': 1, 'u': 1, 'e': 2, 'c': 1}
```

H.W1: Write a Python program to match key values in two dictionaries
Sample dictionary: {'key1': 1, 'key2': 3, 'key3': 2}, {'key1': 1, 'key2': 2}
Expected output: key1: 1 is present in both x and y

H.W2: Write a Python program to replace dictionary values with their sum.
Sample dictionary: {'key1': 1, 'key2': 3, 'key3': 2}
Expected output: {'key1': 6, 'key2': 6, 'key3': 6}

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

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

Defining a Function

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

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon : and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

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

Example

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

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Calling a Function

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

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

```
#!/usr/bin/python

# 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
```

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 −

```
#!/usr/bin/python

# 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]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   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
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result:

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

Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments

- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
#!/usr/bin/python

# 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()
```

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

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme* function in the following ways −

```
#!/usr/bin/python

# 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( str = "My string")
```

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

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
```

```
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

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

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

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

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk * is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;
```

```
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

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

```
Output is:
10
Output is:
70
60
50
```

The *Anonymous* Functions

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.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

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

Following is the example to show how *lambda* form of function works −

```
#!/usr/bin/python

# 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 −

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

```
# 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
```

Scope of Variables

All 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

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 −

```
#!/usr/bin/python

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
```

Examples Sheet #8 (Python Functions)

Exercises with Solutions

1- Write a Python function to find the Max of three numbers.

```
#Solution:
def max_of_two( x, y ):
    if x > y:
        return x
    return y
def max_of_three( x, y, z ):
    return max_of_two( x, max_of_two( y, z ) )
print(max_of_three(3, 6, -5))
#Output:6
```

2- Write a Python function to sum all the numbers in a list.
Sample set : (8, 2, 3, 0, 7)
Output : 20

```
#Solution:
def sum(numbers):
    total = 0
    for x in numbers:
        total += x
    return total
print(sum((8, 2, 3, 0, 7)))
#Output:20
```

3-Write a Python function to reverse a string.
Sample String : "1234abcd"
Output : "dcba4321"

```
#Solution:
def string_reverse(str1):
    rstr1 = ''
    index = len(str1)
    while index > 0:
        rstr1 += str1[ index - 1 ]
        index = index - 1
    return rstr1
print(string_reverse('1234abcd'))
#Output: "dcba4321"
```

4- Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument

```
#Solution:
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

```
n=int(input("Input a number to compute the factiorial : "))
print(factorial(n))
#Output: Input a number to compute the factiorial : 4
#24
```

5- Write a Python function to check whether a number is in a given range.

```
#Solution:
def test_range(n):
    if n in range(3,9):
        print( " %s is in the range"%str(n))
    else :
        print("The number is outside the given range.")
test_range(5)
#Output: 5 is in the range
```

H.W: Write a Python function to print the even numbers from a given list.

Sample List : [1,2,3,4,5,6,7,8,9]
Output : 2 4 6 8

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

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

Example

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

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

The *import* Statement

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

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

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

```
#!/usr/bin/python

# Import module support
import support

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

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

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

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

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

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

```
from fib import fibonacci
```

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

The *from...import* * Statement:

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

```
from modname import *
```

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

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.

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
```

Namespaces and Scoping

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

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

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

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

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

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

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

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

```
Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

The dir Function

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

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

```
#!/usr/bin/python

# 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.

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.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys function.

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)
```

Packages in Python

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

Consider a file *Pots.py* available in *Phone* directory. This file has following line of source code −

```
#!/usr/bin/python

def Pots():
   print "I'm Pots Phone"
```

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

- *Phone/Isdn.py* file having function Isdn
- *Phone/G3.py* file having function G3

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

- Phone/__init__.py

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

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

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

```
#!/usr/bin/python

# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

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

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

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

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

Printing to the Screen

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

```
#!/usr/bin/python

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

This produces the following result on your standard screen −

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

Reading Keyboard Input

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

- raw_input
- input

The *raw_input* Function

The *raw_input[prompt]* function reads one line from standard input and returns it as a string *removingthetrailingnewline*.

```
#!/usr/bin/python

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.

```
#!/usr/bin/python

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)]
Recieved input is :  [10, 20, 30, 40]
```

Opening and Closing Files

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

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

The *open* Function

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

Syntax

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

Here are parameter details:

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read *r*.
- **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*defaultbehavior*.

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

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
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.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
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.
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.
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.
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.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if

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

The *file* Object Attributes

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

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

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.
file.softspace	Returns false if space explicitly required with print, true otherwise.

Example

```
#!/usr/bin/python

# 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

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name

# Close opend file
fo.close()
```

This produces the following result −

```
Name of the file:  foo.txt
```

Reading and Writing Files

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read* and *write* methods to read and write files.

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.

Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opend 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!!
```

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 above.

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10);
print "Read String is : ", str
# Close opend file
fo.close()
```

This produces the following result –

```
Read String is :  Python is
```

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 *seekoffset[, 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.

```
#!/usr/bin/python

# 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*:

```
#!/usr/bin/python
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* −

```
#!/usr/bin/python
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 −

```
#!/usr/bin/python
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 −

```
#!/usr/bin/python
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 −

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

```
#!/usr/bin/python
import os

# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and

manipulate files & directories on Windows and Unix operating systems. They are as follows −

- File Object Methods: The *file* object provides functions to manipulate files.
- OS Object Methods: This provides methods to process files as well as directories.

Examples Sheet #9 (Python FILES I/O)

Exercises with Solutions

1- Write a Python program to read an entire text file text.txt stored in E partition. For example, let the content of the text.txt is:

"What is Python language?
Python is a widely used high-level, general-purpose, interpreted, dynamic programming language."

```
#Solution:
def file_read(fname):
        txt = open(fname)
        print(txt.read())

file_read('E:/text.txt')
#Output:
"""
What is Python language?
Python is a widely used high-level, general-purpose, interpreted, dynamic
programming language.
"""
```

2- Write a Python program to append text to file text.txt and display the text.

```
#Solution:
def file_read(fname):
        from itertools import islice
        with open(fname, "a") as myfile:
                myfile.write("Python Exercises")
        txt = open(fname)
        print(txt.read())
file_read('E:/text.txt')
#Output:
"""
What is Python language?
Python is a widely used high-level, general-purpose, interpreted, dynamic
programming language.
Python Exercises
"""
#Note: if you use <open(fname, "w") as myfile> then the new text will
#overwrite the old one and the output will just be as below:
#Python Exercises
```

3- Write a Python program to read a file line by line and store it into a list.

```
#Solution:
def file_read(fname):
        with open(fname) as f:
                #Content_list is the list that contains the read lines.
                content_list = f.readlines()
                print(content_list)
file_read('E:/text.txt')
#Output:
```

```
['What is Python language?                                                \n',
'Python is a widely used high-level, general-purpose, interpreted, dynamic
programming language.\n', 'Python Exercises']
```

4-Write a Python program to count the number of lines in a text file.

```
#Solution:
def file_lengthy(fname):
    with open(fname) as f:
            for i, l in enumerate(f):
                    pass
    return i + 1
print("Number of lines in the file: ",file_lengthy('E:/text.txt'))
#Output: ('Number of lines in the file: ', 4)
```

5-Write a Python program to write a list to a file.

```
#Solution:
color = ['Red', 'Green', 'White', 'Black', 'Pink', 'Yellow']
with open('E:/abc.txt', "w") as myfile:
        for c in color:
                myfile.write("%s\n" % c)
content = open('E:/abc.txt')
print(content.read())
#Output:
"""
Red
Green
White
Black
Pink
Yellow
"""
```

6-Write a Python program to copy the contents of a file to another file

```
#Solution: from shutil import copyfile; copyfile('E:/abc.txt', 'E:/abc1.txt')
```

NATURAL LANGUAGE PROCESSING

- A natural language is human spoken language, such as English. NLP is a subfield of Artificial Intelligence and linguistics. It studies the problems of automated generation and understanding of natural human languages. Natural language generation systems convert information from computer databases into normal-sounding human language, and natural language understanding systems Convert samples of human language into more formal representations that are easier for computer programs to manipulate.

NATURAL LANGUAGE PROCESSING:

- 1. Understanding written text (written programs, search on internet, Email & chat, Microsoft word,etc).
- 2. understanding spoken language(commands, robotics S/W & H/W, modern car's,…etc)
- 3. signs (describing objects)

NLP GOAL

- The basic goal of (NLP) is to enable a person to communicate with a computer in a Language that they use in their everyday life.

STAGE OF LANGUAGE ANALYSIS:

- **Generally Five Processing Stages in a NLP System**
- **Phonological Analysis**
- **Morphological Analysis (lexical)**
- **Syntactic Analysis**
- **Semantic Analysis**
- **Pragmatic Analysis**

WRITTEN TEXT PROCESSING:

- 1. Formal method.
- 2. Informal method

- **Informal method:**
- Example:
- Computers milk drinks.
- Computer drinks milk.
- Computers use data.

- **Formal method:**
- 1. lexical analysis.(word)
- 2. syntactical analysis.(grammars)
- 3. semantic analysis.(meanings)

INTELLIGENT ROBOT:

- The robot would have to know:
- 1. The meaning of the words.
- 2. Relationship of one word to another.
- 3. Knowledge of grammars.
- 4. Associate descriptions and objects.
- 5. Analyze sentence in relation to another sentences.

- e.g
- - John drank milk
- -he then put on his coat.

WHAT UNDERSTANDS?

- To understand something is transform it from one representation into another, where this second representation has been chosen to correspond to a set of available actions that could be performed and where the mapping has been designed. So that for each event, an appropriate action will be done.

STAGE OF UNDERSTANDING NLP:

- Generally Processing Stages in a NLP System

1. Phonological Analysis
2. Morphological Analysis (lexical Analysis)
3. Syntactic Analysis
4. Semantic Analysis
5. Discourse analysis
6. Pragmatic Analysis

THE PHONOLOGICAL ANALYSIS

- Analysis of speech sounds of the world's languages, with a focus on both their articulator and acoustic properties. An introduction to phonetic alphabets, including practice in transcribing a variety of language samples. Analysis of the systematic organization of speech sounds, with reference to features and supra segmental.

THE LEXICAL ANALYZER

- Reads the input text of source language character and produces tokens such as (*names , keywords, punctuation marks ,discards white space and comments), which are* the basic lexical units of the language. The process of breaking-up a text into its constituent tokens is known as tokenization. Tokenization occurs at a number of different levels: a text could be broken up into paragraphs, sentences, words, syllables, or phonemes.

SYNTACTIC ANALYSIS

- Is concerned with the construction of sentences. Syntactic structure indicates how the words are related to each other. Syntax tree is assigned by a grammar and a lexicon.

SEMANTIC ANALYSIS

- Which produce a representation of the meaning of the text, the representation that commonly used include conceptual dependency, frames, and logic base representation. and it uses the knowledge about the meaning of the word and linguist structure, such as case roles of nouns or the transitivity.

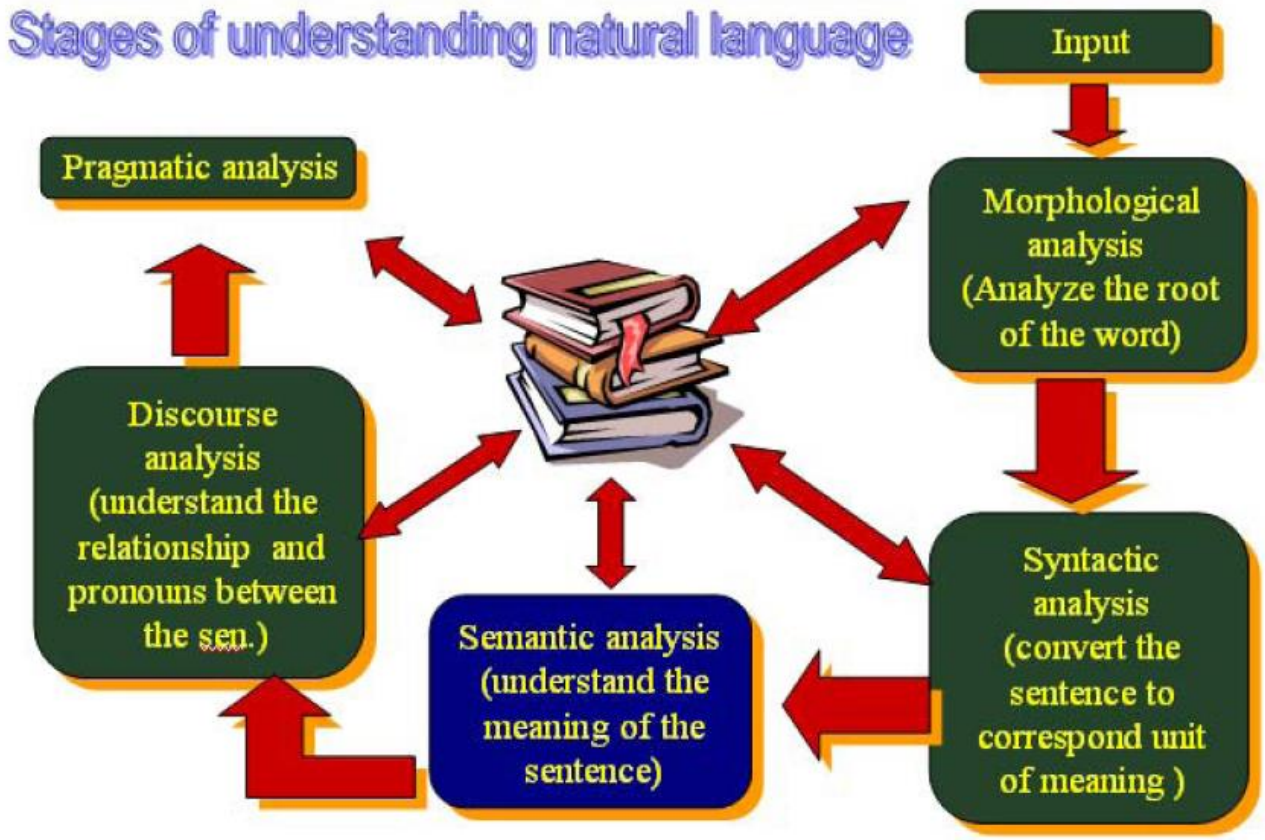
DISCOURSE ANALYSIS

- Analyze the effect of previous sentence on the next sentence.

PRAGMATIC ANALYSIS

- Interpret the sentence according to its meaning.

STAGES OF UNDERSTANDING NL



References:

1. tutorials point simply easy learning, " Python programming language" , copyrighted 2014. https://www.tutorialspoint.com/python/index.htm
2. S. Bird, E. Klein, and E. Loper (2009)"Natural Language Processing with Python"
3. D. Jurafsky and J. Martin (2009) "SPEECH and LANGUAGE PROCESSING-An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition