

# Chapter-1

## INTRODUCTION

**Overview:** Python is a high-level, interpreted, interactive, object-oriented, general purpose scripting language. Python is designed to be highly readable.

**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 December 1989) 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.

### Who uses PYTHON ?

#### Global Players

Google  
YouTube  
Dropbox  
Intel/Cisco/HP  
NASA

#### Local Players

HCL Technologies  
MaYo Technologies  
Snapdeal

### What can you do with PYTHON ?

System Programming	Internet Scripting
Database programming – SQL, “BIG DATA”	Rapid Prototyping
Numeric and scientific programming	Gaming
MATLAB	
GUI Programming	

### Technical Strengths

Object oriented.	Dynamic typing
It is free	Automatic memory management

It is Portable.  
Powerful

Built in object type – list, dictionary,  
Libraries, third party utilities.

**Getting Python** : The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python:  
**<http://www.python.org/>**

Python Documentation Website : **[www.python.org/doc/](http://www.python.org/doc/)**

**Install Python:** Follow this link (<http://www.python.org/download/>) for installation on various operating system such as window linux, mac etc.

**Setting up PATH** The path variable is named as PATH in Unix or Path in Windows (Unix is case-sensitive; Windows is not).

**Setting path at Unix/Linux:** To add the Python directory to the path for a particular session in **Unix**:

**In the csh shell:** type **setenv PATH "\$PATH:/usr/local/bin/python"** and press Enter.

**In the bash shell (Linux):** type **export PATH="\$PATH:/usr/local/bin/python"** and press Enter.

**In the sh or ksh shell:** type **PATH="\$PATH:/usr/local/bin/python"** and press Enter.

**Note:** /usr/local/bin/python is the path of the Python directory

**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

## **Integrated Development Environment(IDE)**

- 1.IDLE
- 2.Spyder
3. PyCharm

## **PYTHON BASIC SYNTAX**

### **Python Identifiers**

A Python identifier is a name used to identify a variable, function, class, module or other object. Python is a case sensitive programming language. 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.

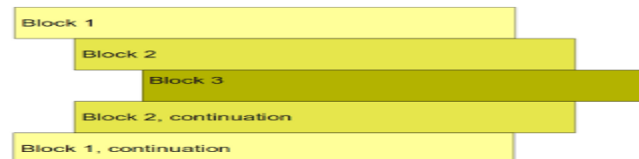
**Reserved Words:** All the Python keywords contain lowercase letters only.

<i>And</i>	<i>Exec</i>	<i>Not</i>	<i>Except</i>	<i>Lambda</i>	<i>yield</i>	<i>Else</i>	<i>is</i>	<i>With</i>
<i>Assert</i>	<i>Finally</i>	<i>Or</i>	<i>Def</i>	<i>If</i>	<i>return</i>	<i>Continue</i>	<i>Global</i>	<i>Raise</i>
<i>Break</i>	<i>For</i>	<i>Pass</i>	<i>Del</i>	<i>Import</i>	<i>try</i>	<i>Elif</i>	<i>in</i>	<i>While</i>
<i>Class</i>	<i>From</i>	<i>Print</i>						

## 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.

```
begin ... end
do ... done
{ ... }
if ... fi
```



```
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"
```

## 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 ("" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

```
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.

### Python Variable Types

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

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.

```
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.

*Example1:*

***a = b = c =1***

*Example2:*

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

## Chapter-2

# STANDARD DATA TYPES

The data stored in memory can be of many types. Python has five standard data types:

- **1.Numbers**
- **2.String**
- **3.List**
- **4.Tuple**
- **5.Dictionary**

### Python Numbers

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.

Python supports different numerical types

- `int()`
- `float()`
- `complex(x,y)`

### Mathematical Functions

Python includes following functions that perform mathematical calculations and all these functions cannot be used directly and for that we need to import the math function in it.

Example-

Function	What it'll Return ( description )
<code>abs(x)</code>	The absolute value of x: the (positive) distance between x and zero.
<code>ceil(x)</code>	The ceiling of x: the smallest integer not less than x
<code>cmp(x, y)</code>	-1 if $x < y$ , 0 if $x == y$ , or 1 if $x > y$
<code>exp(x)</code>	The exponential of x: $e^x$

<code>fabs(x)</code>	The absolute value of x.
<code>floor(x)</code>	The floor of x: the largest integer not greater than x
<code>log(x)</code>	The natural logarithm of x, for $x > 0$
<code>log10(x)</code>	The base-10 logarithm of x for $x > 0$ .
<code>max(x1, x2,...)</code>	The largest of its arguments: the value closest to positive infinity
<code>min(x1, x2,...)</code>	The smallest of its arguments: the value closest to negative infinity
<code>modf(x)</code>	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.
<code>pow(x, y)</code>	The value of $x^{**}y$ .
<code>round(x [,n])</code>	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: <code>round(0.5)</code> is 1.0 and <code>round(-0.5)</code> is -1.0.
<code>sqrt(x)</code>	The square root of x for $x > 0$

## Random Number Functions

Function	Description
<code>choice(seq)</code>	A random item from a list, tuple, or string.

<b>randrange ([start,] stop [,step])</b>	A randomly selected element from range(start, stop, step)
<b>random()</b>	A random float r, such that 0 is less than or equal to r and r is less than 1
<b>shuffle(lst)</b>	Randomizes the items of a list in place. Returns None.

## 1. Choice()

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

When we run above program, it produces following result:

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

## 2. randrange()

```
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
```



### 3. random()

The method **random()** returns a random float *r*, such that 0 is less than or equal to *r* and *r* is less than 1.

```
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
```

### 4 shuffle()

The method **shuffle()** randomizes the items of a list in place.

```
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]
```

**STRINGS** – They are bunch of characters. In other words they are identified as contiguous(IN SEQUENCE) set of characters. Subset of strings can be taken using slice operator [],[:]

## Accessing Values in Strings

```
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

```
var1 = 'Hello World!'
print"Updated String :- ", var1[:6]+'Python'
```

## String Formatting Operator

One of Python's coolest features is the string format operator %. Following is a simple example –

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

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

```
My name is Mac 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
%d	signed decimal integer
%f	floating point real number

## Built-in String Methods

### 1. Capitalize() Method

It returns a copy of the string with only its first character capitalized.

#### Syntax

```
str.capitalize()
```

#### Example

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

#### Result

```
str.capitalize():Thisisstring example....wow!!!
```

### 2. Count() Method

#### Description

The method **count()** returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

## Syntax

```
str.count(sub, start, end)
```

## Parameters

- **sub** -- This is the substring to be searched.
- **start** -- Search starts from this index. First character starts from 0 index. By default search starts from 0 index.
- **end** -- Search ends from this index. First character starts from 0 index. By default search ends at the last index.

## Return Value

Centered in a string of length width.

## Example

```
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, 4, 40) : 1
```

### 3. **Decode() Method**

The method **decode()** decodes the string using the codec registered for *encoding*. It defaults to the default string encoding.

## Syntax

```
Str.decode(encoding='UTF-8', errors='strict')
```

## Parameters

- **encoding** -- This is the encodings to be used. For a list of all encoding schemes please visit: [Standard Encodings](#).
- **errors** -- This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register\_error()..

## Return Value

Decoded string.

## Example

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

## Result

```
EncodedString: dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE=
DecodedString: thisisstring example....wow!!!
```

## 4. Find() Method

### Description

It determines if string *str* occurs in string, or in a substring of string if starting index *beg* and ending index *end* are given.

### Syntax

```
str.find(str)
```

### Parameters

- **str** -- This specifies the string to be searched.
- **beg** -- This is the starting index, by default its 0.

- **end** -- This is the ending index, by default its equal to the lenght of the string.

## Return Value

Index if found and -1 otherwise.

## Example

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

## Result

15

## 5. index() Method

### Description

It determines if string *str* occurs in string or in a substring of string if starting index *beg* and ending index *end* are given. This method is same as *find()*, but raises an exception if sub is not found.

### Syntax

```
str.index(str, beg=0end=len(string))
```

### Parameters

- **str** -- This specifies the string to be searched.
- **beg** -- This is the starting index, by default its 0.
- **end** -- This is the ending index, by default its equal to the length of the string.

## Return Value

Index if found otherwise raises an exception if str is not found.

## Example

```
str1 = "this is string example...wow!!!"  
str2 = "exam"
```

```
print str1.index(str2)
print str1.index(str2,10)
print str1.index(str2,40)
```

## 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
```

## 6. isalnum() Method

### Description

The method **isalnum()** checks whether the string consists of alphanumeric characters.

### Return Value

This method returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

### Example

The following example shows the usage of **isalnum()** method.

```
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
```

## 7. isalpha() Method

### Description

The method **isalpha()** checks whether the string consists of alphabetic characters only.

### Syntax

Following is the syntax for **isalpha()** method –

```
str.isalpha()
```

### Example

The following example shows the usage of isalpha() method.

```
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
```

## 8. islower() Method()

### Description

The method **islower()** checks whether all the case-based characters (letters) of the string are lowercase.

### Syntax

Following is the syntax for **islower()** method –

```
str.islower()
```



## Example

The following example shows the usage of `islower()` method.

```
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
```

## 9. join() Method

### Description

The method **join()** returns a string in which the string elements of sequence have been joined by *str* separator.

### Syntax

Following is the syntax for **join()** method:

```
str.join(sequence)
```

## Example

The following example shows the usage of `join()` method.

```
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
```

## 10. len() Method

## Description

The method **len()** returns the length of the string.

## Syntax

Following is the syntax for **len()** method –

```
len( str )
```

## Example

The following example shows the usage of **len()** method.

```
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
```

## 11. ljust() Method

### Description

The method **ljust()** returns the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if width is less than **len(s)**.

### Syntax

Following is the syntax for **ljust()** method –

```
str.ljust(width[, fillchar])
```

### Parameters

- **width** -- This is string length in total after padding.
- **fillchar** -- This is filler character, default is a space.

## Return Value

This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

## Example

The following example shows the usage of ljust() method.

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

When we run above program, it produces following result –

```
thisisstring example....wow!!!000000000000000000
```

## 12. lower() Method

### Description

The method **lower()** returns a copy of the string in which all case-based characters have been lowercased.

### Syntax

Following is the syntax for **lower()** method –

```
str.lower()
```

## Example

The following example shows the usage of lower() method.

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

When we run above program, it produces following result –

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

### 13. lstrip() Method.

#### Description

The method **lstrip()** returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

#### Syntax

Following is the syntax for **lstrip()** method –

```
str.lstrip([chars])
```

#### Example

The following example shows the usage of **lstrip()** method.

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

When we run above program, it produces following result –

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

### 14. **rstrip()** Method

#### Description

The method **rstrip()** returns a copy of the string in which all *chars* have been stripped from the end of the string (default whitespace characters).

#### Syntax

Following is the syntax for **rstrip()** method –

```
str.rstrip([chars])
```

## Example

The following example shows the usage of `rstrip()` method.

```
#!/usr/bin/python

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

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

When we run above program, it produces following result –

```
thisisstring example....wow!!!
88888888thisisstring example....wow!!!
```

## 15. `rstrip()` Method

### Description

The method **`strip()`** returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

### Syntax

Following is the syntax for **`strip()`** method –

```
str.strip([chars])
```

## Example

The following example shows the usage of `strip()` method.

```
#!/usr/bin/python

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!!!
```

## 16. maketrans() Method

### Description

The method **maketrans()** returns a translation table that maps each character in the *intabstring* into the character at the same position in the *outtab* string. Then this table is passed to the `translate()` function.

**Note:** Both *intab* and *outtab* must have the same length.

### Syntax

Following is the syntax for **maketrans()** method –

```
str.maketrans(intab, outtab])
```

### Parameters

- **intab** -- This is the string having actual characters.
- **outtab** -- This is the string having corresponding mapping character.

### Example

The following example shows the usage of `maketrans()` method. Under this, every vowel in a string is replaced by its vowel position –

```
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 2x1mp12...w4w!!!
```

### **17. *max()* Method**

#### **Description**

The method **max()** returns the max alphabetical character from the string *str*.

#### **Syntax**

Following is the syntax for **max()** method –

```
max(str)
```

#### **Example**

The following example shows the usage of **max()** method.

```
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
```

### **18. *min()* Method**

#### **Description**

The method **min()** returns the min alphabetical character from the string *str*.

#### **Syntax**

Following is the syntax for **min()** method:

```
min(str)
```

#### **Example**

The following example shows the usage of min() method.

```
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: !
```

### **19.replace() method**

#### **Description**

The method **replace()** returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

#### **Syntax**

Following is the syntax for **replace()** method –

```
str.replace(old,new[, max])
```

#### **Parameters**

- **old** -- This is old substring to be replaced.
- **new** -- This is new substring, which would replace old substring.
- **max** -- If this optional argument max is given, only the first count occurrences are replaced.

#### **Return Value**

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

#### **Example**

The following example shows the usage of replace() method.



```
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
```

## **20. split() Method**

### **Description**

The method **split()** returns a list of all the words in the string, using *str* as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to *num*.

### **Example**

The following example shows the usage of split() method.

```
str ="black,color"  
print str.split()  
print str.split(',')
```

When we run above program, it produces following result –

```
['black,color']  
['black','color']
```

## **21. upper() Method**

### **Description**

The method **upper()** returns a copy of the string in which all case-based characters have been uppercased.

### **Syntax**

Following is the syntax for **upper()** method –

```
str.upper()
```

## Example

The following example shows the usage of upper() method.

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

When we run above program, it produces following result –

```
THIS IS STRING EXAMPLE....WOW!!!
```

## LISTS :

List are the most versatile (ABLE TO ADAPT) of python. Lists contain items separated by comma and enclosed within square brackets []. Lists are similar to arrays in C. They are used for storing all data types. ex- int, float, string, etc.

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

```
list1 = ['Lucy', 'Avergers', 1997, 2000] \
```

## Accessing Values in Lists

```
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

```
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]
```

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

```
list1 = ['physics', 'chemistry', 1997, 2000]
print list1
del list1[2]
print "After deleting value at index 2 : "
print list1
```

## 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
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	1 2 3	Iteration

## Python List Builtin Methods

### 1. `append()` Method

#### Description

The method **`append()`** appends a passed *obj* into the existing list.

#### Syntax

Following is the syntax for **`append()`** method –

```
list.append(obj)
```

#### Example

The following example shows the usage of `append()` method.

```
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]
```

## 2. Python List count() Method

### Description

The method **count()** returns count of how many times *obj* occurs in list.

### Syntax

Following is the syntax for **count()** method –

```
list.count(obj)
```

### Example

The following example shows the usage of count() method.

```
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
```

## 3. Python List extend() Method

### Description

The method **extend()** appends the contents of *seq* to list.

### Syntax

Following is the syntax for **extend()** method –

```
list.extend(seq)
```

## Example

The following example shows the usage of `extend()` method.

```
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']
```

## 4. Python List `index()` Method

### Description

The method **`index()`** returns the lowest index in list that *obj* appears.

### Syntax

Following is the syntax for **`index()`** method –

```
list.index(obj)
```

## Example

The following example shows the usage of `index()` method.

```
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

## 5. Python List insert() Method

### Description

The method **insert()** inserts object *obj* into list at offset *index*.

### Syntax

Following is the syntax for **insert()** method –

```
list.insert(index, obj)
```

### Parameters

- **index** -- This is the Index where the object *obj* need to be inserted.
- **obj** -- This is the Object to be inserted into the given list.

### Return Value

This method does not return any value but it inserts the given element at the given index.

### Example

The following example shows the usage of **insert()** method.

```
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']
```

## 6. Python List pop() Method

### Description

The method **pop()** removes and returns last object or *obj* from the list.

### Syntax

Following is the syntax for **pop()** method –

```
list.pop(obj=list[-1])
```

### Parameters

- **obj** -- This is an optional parameter, index of the object to be removed from the list.

### Return Value

This method returns the removed object from the list.

### Example

The following example shows the usage of pop() method.

```
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
```

## 7. Python List remove() Method

### Parameters

- **obj** -- This is the object to be removed from the list.

### Return Value

This method does not return any value but removes the given object from the list.



## Example

The following example shows the usage of `remove()` method.

```
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']
```

## 8. Python List `reverse()` Method

### Description

The method **`reverse()`** reverses objects of list in place.

### Syntax

Following is the syntax for **`reverse()`** method –

```
list.reverse()
```

## Example

The following example shows the usage of `reverse()` method.

```
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]
```

## 9. Python List sort() Method

### Description

The method **sort()** sorts objects of list, use compare *func* if given.

### Syntax

Following is the syntax for **sort()** method –

```
list.sort([func])
```

### Example

The following example shows the usage of sort() method.

```
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']
```

## TUPLES :

They are just like list. They cannot be adjusted. We use () in tUPLES. It is a sequence of immutable(UNABLE TO CHANGE) objects. Value can't be deleted or updated in it.

TUPLES VS LIST :

- |  |                            |
|--|----------------------------|
| 1. More memory efficient                                   | 1. Takes more memory       |
| 2. Cannot be adjusted.Means nothing can be changed updated | 2. It's adjustable. Can be |

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)
```

## Accessing Values in Tuples:

```
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

```
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

```
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

## Built-in Tuple Functions

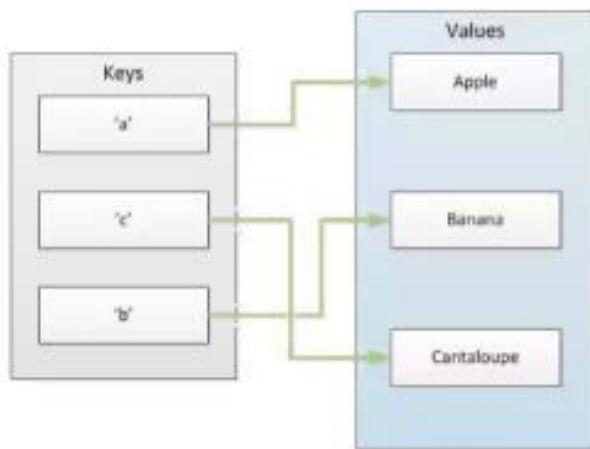
Python includes the following tuple functions –

SN	Function with Description
1	<b>cmp(tuple1, tuple2)</b>  Compares elements of both tuples.
2	<b>len(tuple)</b>  Gives the total length of the tuple.
3	

	<b>max(tuple)</b>
	Returns item from the tuple with max value.
4	<b>min(tuple)</b>
	Returns item from the tuple with min value.
5	<b>tuple(seq)</b>
	Converts a list into tuple.

## Python Dictionary

**Python** dictionary is a data structure similar to an associative array found in other programming languages. An array or a list indexes elements by position. A dictionary, on the other hand, indexes elements by **keys** which can be strings. Think of a dictionary as unordered sets of **key-value** pairs.



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.

## Accessing Values in Dictionary:

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

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

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

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

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

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

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

## Updating Dictionary

```
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

```
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
```

## Nested Dictionaries Examples

```
DICTIONARY IN DICTIONARY  
  
dict1={'Name':{'first':'will','last':'smith'},'Age':'17','occup':['Teaching',  
'supervisor','manager']}  
  
print dict1['Name']  
print dict1['Name']['last']  
print dict1['Name']['first']  
print (dict1['occup'][1])  
print (dict1['occup'][0])  
  
dict1['occup'].append('VP')  
print dict1 ["occup"]
```



## Built-in Dictionary Functions

### 1. Python dictionary copy() Method

#### Description

The method **copy()** returns a shallow copy of the dictionary.

#### Syntax

Following is the syntax for **copy()** method –

```
dict.copy()
```

#### Example

The following example shows the usage of copy() method.

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

When we run above program, it produces following result –

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

### 2. Python dictionary fromkeys() Method

#### Description

The method **fromkeys()** creates a new dictionary with keys from *seq* and *values* set to value.

#### Syntax

Following is the syntax for **fromkeys()** method –

```
dict.fromkeys(seq[, value])
```

#### Example

The following example shows the usage of fromkeys() method.

```
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}
```

### 3. Python dictionary `has_key()` Method

#### Description

The method **`has_key()`** returns true if a given key is available in the dictionary, otherwise it returns a false.

#### Syntax

Following is the syntax for **`has_key()`** method –

```
dict.has_key(key)
```

#### Example

The following example shows the usage of `has_key()` method.

```
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
```

### 4. Python dictionary `keys()` Method

## Description

The method **keys()** returns a list of all the available keys in the dictionary.

## Syntax

Following is the syntax for **keys()** method –

```
dict.keys()
```

## Example

The following example shows the usage of keys() method.

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

When we run above program, it produces following result –

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

## 5. Python dictionary values() Method

### Description

The method **values()** returns a list of all the values available in a given dictionary.

### Syntax

Following is the syntax for **values()** method –

```
dict.values()
```

### Example

The following example shows the usage of values() method.

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

When we run above program, it produces following result –

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

## 6. Python dictionary update() Method

### Description

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

### Syntax

Following is the syntax for **update()** method –

```
dict.update(dict2)
```

### Example

The following example shows the usage of update() method.

```
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'}
```

## Dictionary as a set of counters

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
```

```
print d
```

The for loop traverses the string. Each time through the loop, if the character *c* is not in the dictionary, we create a new item with key *c* and the initial value 1 (since we have seen this letter once). If *c* is already in the dictionary we increment *d[c]*.

Here's the output of the program:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

## Looping and dictionaries

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}  
for key in counts:  
    print key, counts[key]
```

Here's what the output looks like:

```
jan 100  
chuck 1  
annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}  
for key in counts:  
    if counts[key] > 10 :  
        print key, counts[key]
```

The for loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here's what the output looks like:

```
jan 100  
annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the `keys` method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key printing out key/value pairs in sorted order as follows:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
    print key, counts[key]
```

Here's what the output looks like:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

First you see the list of keys in unsorted order that we get from the `keys` method. Then we see the key/value pairs in order from the for loop.

## Specifying Key-Value Pairs

You can also create and initialize a dictionary using name value pairs as keyword arguments to the **`dict()`** constructor.

```
p=dict(california=33,Colorado=46356)
print p
```

Output:

```
{'california': 33, 'Colorado': 46356}
```

## Array of Key-Value Tuples

Yet another way of creating a dictionary is to use an array of key-value tuples. Here is the same example as above

```
pairs = [('California', 37253956), ('Colorado', 56)]  
print dict(pairs)
```

Output:

```
{'California': 37253956, 'Colorado': 56}
```

## Dict Comprehension

```
print {x: x**2 for x in range(10, 20)}
```

Output:

```
{10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225, 16: 256, 17: 289, 18: 324, 19: 361}
```

How does it work? The latter part (**for x in range(10, 20)**) returns a range of numbers in the specified range. The dict comprehension part (**{x: x\*\*2 ..}**) loops over this range and initializes the dictionary.

## Checking for Existence

What if you want to check for the presence of a key without actually attempting to access it (and possibly encountering a **KeyError** as above)? You can use the **in** keyword in the form **key in dct** which returns a boolean

```
dic={'Name':'Aman','Age':'17','occup':'Teaching'}  
print 'Name' in dic  
print 'name' in dic
```

Output:

```
True
```

```
False
```

## Deleting elements

Use the **pop()** method instead, when you want the deleted value back

```
dic={'Name':'Aman','Age':'17','occup':'Teaching'}  
print dic.pop('Age')  
print dic
```

Output:

```
17  
{'Name': 'Aman', 'occup': 'Teaching'}
```



## Chapter-3

# PYTHON BASIC OPERATORS

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

### Types of Operator

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

### ARITHMETIC OPERATORS

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 $9.0//2.0 = 4.0$

Example:

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

```
a=21
```

```
b=10
```

```
c=0
```

```
c = a + b
```

```
print"Line 1 - Value of c is ", c
```

```
c = a - b
```

```
print"Line 2 - Value of c is ", c
```

```
c = a * b
```

```
print"Line 3 - Value of c is ", c
```

```
c = a / b
```

```
print"Line 4 - Value of c is ", c
```

```
c = a % b
```

```
print"Line 5 - Value of c is ", c
```

```
a=2
```

```
b=3
```

```
c = a**b
```

```
print"Line 6 - Value of c is ", c
```

```
a=10
```

```
b=5
```

```
c = a//b
```

```
print"Line 7 - Value of c is ", c
```

Output:

Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 8

Line 7 - Value of c is 2

## Python Comparison Operators

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.
----	--	-------------------

### Example:

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

```
a =21
```

```
b =10
```

```
c =0
```

```
if( a == b ):
```

```
print"Line 1 - a is equal to b"
```

```
else:
```

```
print"Line 1 - a is not equal to b"
```

```
if( a != b ):
```

```
print"Line 2 - a is not equal to b"
```

```
else:
```

```
print"Line 2 - a is equal to b"
```

```
if( a <> b ):
```

```
print"Line 3 - a is not equal to b"
```

```
else:
```

```
print"Line 3 - a is equal to b"
```

```
if( a < b ):
```

```
print"Line 4 - a is less than b"
```

```
else:
```

```
print"Line 4 - a is not less than b"
```

```
if( a > b ):
```

```
print"Line 5 - a is greater than b"
```

```
else:
```

```
print"Line 5 - a is not greater than b"
```

```
a =5;
```

```

b =20;
if( a <= b ):
print"Line 6 - a is either less than or equal to b"
else:
print"Line 6 - a is neither less than nor equal to b"
if( b >= a ):
print"Line 7 - b is either greater than or equal to b"
else:
print"Line 7 - b is neither greater than nor equal to b"

```

Output:

```

Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not equal to b
Line 4 - a is not less than b
Line 5 - a is greater than b
Line 6 - a is either less than or equal to b
Line 7 - b is either greater than or equal to b

```

## Python Assignment Operators

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	It multiplies right operand with the left operand and	c *= a is equivalent to c

AND	assign the result to left operand	$= c * a$
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	$c /= a$ is equivalent to $c = c / a$ $c /= a$ is equivalent to $c = c / a$
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	$c \% = a$ is equivalent to $c = c \% a$
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	$c ** = a$ is equivalent to $c = c ** a$
//= Floor Division	It performs floor division on operators and assign value to the left operand	$c //= a$ is equivalent to $c = c // a$

### Example

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

```
a = 21
```

```
b = 10
```

```
c = 0
```

```
c = a + b
```

```
print "Line 1 - Value of c is ", c
```

```
c += a
```

```
print "Line 2 - Value of c is ", c
```

```
c *= a
```

```
print "Line 3 - Value of c is ", c
```

```
c /= a
```

```
print "Line 4 - Value of c is ", c
```

```
c = 2
```

```
c %= a
```

```
print "Line 5 - Value of c is ", c
```

```

c**= a
print"Line 6 - Value of c is ", c
c/= a
print"Line 7 - Value of c is ", c

```

Example:

```

Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864

```

## Python Logical Operators:

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

## 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.

**Example:**

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

```
a =10
```

```
b =20
```

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

```
if( a in list):
```

```
print"Line 1 - a is available in the given list"
```

```
else:
```

```
print"Line 1 - a is not available in the given list"
```

```
if( b notin list):
```

```
print"Line 2 - b is not available in the given list"
```

```
else:
```

```
print"Line 2 - b is available in the given list"
```

```
a =2
```

```
if( a in list):
```

```
print"Line 3 - a is available in the given list"
```

```
else:
```

```
print"Line 3 - a is not available in the given list"
```

**Output:**

```
Line 1 - a is not available in the given list
```



Line 2 - b is not available in the given list  
Line 3 - a is available in the given list

## Identity operators

**Identity operators** compare the memory locations of two objects. There are two Identity operators as 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).

## Example

```
#!/usr/bin/python

a =20
b =20

if( a is b ):
    print"Line 1 - a and b have same identity"
else:
```

```
print"Line 1 - a and b do not have same identity"

if( id(a)== id(b)):
    print"Line 2 - a and b have same identity"
else:
    print"Line 2 - a and b do not have same identity"

b =30
if( a is b ):
    print"Line 3 - a and b have same identity"
else:
    print"Line 3 - a and b do not have same identity"

if( a isnot b ):
    print"Line 4 - a and b do not have same identity"
else:
    print"Line 4 - a and b have same identity"
```

When you execute the above program it produces the following result –

```
Line 1 - a and b have same identity
Line 2 - a and b have same identity
Line 3 - a and b do not have same identity
Line 4 - a and b do not have same identity
```

## Chapter-4

# CONDITIONALS AND LOOPING

A way to check for something. One of the example of conditionals is

1. if (CONDITIONAL IS TRUE)statement -] Do this

Example

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

The conditional test passes, and both print statements are indented, so both lines are printed:

You are old enough to vote!  
Have you registered to vote yet?

If the value of age is less than 18, this program would produce no output.

2. else - Now comes the else statement , it works great with the if statement . It is used when anything is passed by the 'if' statement or doesn't meet the criteria Catches everything that does NOT meet prior conditionals.

```
if (....)
    .....
else:
    .....
```

Example

We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
age = 17
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

If the conditional test at u passes, the first block of indented print statements is executed. If the test evaluates to False, the else block at v is executed. Because age is less than 18 this time, the conditional test fails and the code in the else block is executed:

Sorry, you are too young to vote.  
Please register to vote as soon as you turn 18!

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The if-else structure works well in situations in which you want Python to always execute one of two possible actions. In a simple if-else chain like this, one of the two actions will always be executed.

3. elif – Combination of else and if. It comes AFTER if statement means it fits right in b/w else and if. It's used when some other more conditionals are needed to be checked.  
It sets up another conditionals.

Example

```
age = 12
if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $5.")
else:
    print("Your admission cost is $10.")
```

The if test at u tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The elif line at v is really another if test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is less than 18, an appropriate message is printed and Python skips the else block. If both the if and elif tests fail, Python runs the code in the else block at w. In this example the test at u evaluates to False, so its code block is not executed. However, the second test evaluates to True (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

Your admission cost is \$5.

Any age greater than 17 would cause the first two tests to fail. In these situations, the else block would be executed and the admission price would be \$10.

## LOOPS :

Loops are the way to repeat a single action over and over.

Ex : real world example of loop 'll be wash a dish over and over. U wash a single dish then u grab another dish to wash.

### while Loop Statements

1. **while** : (condition should be true)

A while loop implements the repeated execution of code based on a given Boolean condition. The code that is in a while block will execute as long as the while statement evaluates to True.

You can think of the while loop as a repeating conditional statement. After an if statement, the program continues to execute code, but in a while loop, the program jumps back to the start of the while statement until the condition is False.

As opposed to for loops that execute a certain number of times, while loops are conditionally based, so you don't need to know how many times to repeat the code going in.

## Syntax

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

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

## Examples

Let's create a small program that executes a while loop. In this program, we'll ask for the user to input a password. While going through this loop, there are two possible outcomes:

- If the password *is* correct, the while loop will exit.
- If the password is *not* correct, the while loop will continue to execute.

We'll create a file called password.py in our text editor of choice, and begin by initializing the variable password as an empty string:

password.py

```
password = ''
```

The empty string will be used to take in input from the user within the `while` loop.

Now, we'll construct the `while` statement along with its condition:

password.py

```
password = ''  
while password != 'password':
```

Here, the `while` is followed by the variable `password`. We are looking to see if the variable `password` is set to the string `password` (based on the user input later), but you can choose whichever string you'd like.

This means that if the user inputs the string `password`, then the loop will stop and the program will continue to execute any code outside of the loop. However, if the string that the user inputs is not equal to the string `password`, the loop will continue.

Next, we'll add the block of code that does something within the `while` loop:

password.py

```
password = ''

while password != 'password':
    print('What is the password?')
    password = input()
```

Inside of the `while` loop, the program runs a print statement that prompts for the password. Then the variable `password` is set to the user's input with the `input()` function.

The program will check to see if the variable `password` is assigned to the string `password`, and if it is, the `while` loop will end. Let's give the program another line of code for when that happens:

password.py

```
password = ''

while password != 'password':
    print('What is the password?')
    password = input()

print('Yes, the password is ' + password + '. You may enter.')
The last print() statement is outside of the while loop, so when the user enters password as the password, they will see the final print statement outside of the loop.
```

However, if the user never enters the word `password`, they will never get to the last `print()` statement and will be stuck in an infinite loop.

An **infinite loop** occurs when a program keeps executing within one loop, never leaving it. To exit out of infinite loops on the command line, press `CTRL + C`.

Save the program and run it:

You'll be prompted for a password, and then may test it with various possible inputs. Here is sample output from the program:

Output

```
What is the password?
```

```
hello
```

```
What is the password?
```

```
sammy
```

```
What is the password?
```

```
PASSWORD
```

```
What is the password?
```

```
password
```

```
Yes, the password is password. You may enter.
```

Keep in mind that strings are case sensitive unless you also use a string function to convert the string to all lower-case (for example) before checking.

## Example

```
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!
```

## Example

```
#!/usr/bin/python  
  
count= 10  
while(count >0):  
    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: 10  
The count is: 9  
The count is: 8  
The count is: 7  
The count is: 6  
The count is: 5  
The count is: 4  
The count is: 3  
The count is: 2  
The count is: 1  
Good bye!
```

## 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
```

## For Loop Statements

It has ability to iterate over the item of any sequence , such as '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.

### 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.

**range** : now combine the power of for loop with range. A range is a key word function that creates a sequence of numbers and by default it goes from 0 up to, but not including the last value you put in.

```
Ex – range(6)
range(2,7)
range(10,20,2)
```

Remember range takes upper bound one less and lower bound from the same point of initialization.

```
print range(6)           # here 6 is behaving as upper bound
print range(2,7)         # and 2 is behaving as lower bound
print range(10,20,2)
```

## OUTPUT

```
[0,1,2,3,4,5]
[2,3,4,5,6]
[10,12,14,16,18]
```

## 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. Click the following links to check their detail.

Control Statement	Description
<b>break statement</b>	Terminates the loop statement and transfers execution to the statement immediately following the loop.
<b>continue statement</b>	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
<b>pass statement</b>	The pass statement in Python is used when a statement is required syntactically but you do not want any

## Python break statement

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

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

## Syntax

The syntax for a **break** statement in Python is as follows –

```
break
```

## Example

```
#!/usr/bin/python

for letter in 'Python':# First Example
    if letter == 'h':
        break
    print 'Current Letter :', letter

var=10# Second Example
while var>0:
    print 'Current variable value :',var
    var=var-1
    if var==5:
        break

print "Good bye!"
```

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

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
```

```
Current variable value : 8  
Current variable value : 7  
Current variable value : 6  
Good bye!
```

## Python continue statement

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

The **continue** statement can be used in both *while* and *for* loops.

## Syntax

```
continue
```

## Example

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

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

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

## Example

Program to exclude the number which are divisible by 3 till range 30.

```
for i in range:
    if (i%3==0):
        continue# Third Example
    print i
```

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

```
1
2
4
5
7
8
10
11
13
```



14

16

17

19

20

22

23

25

26

28

29

➔ Now try to make similar program with the help of while loop.

## Python pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

## Syntax

```
pass
```

## Example

```
#!/usr/bin/python

for letter in 'Python':
    if letter == 'h':
        pass
```

```
print 'This is pass block'  
print 'Current Letter :', letter  
  
print "Good bye!"
```

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

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

# Chapter-5

## NESTED FOR LOOPS

### Nested For Loops

Loops can be nested in Python, as they can with other programming languages.

A nested loop is a loop that occurs within another loop, structurally similar to nested `if` statements. These are constructed like so:

```
for [first iterating variable] in [outer loop]: # Outer loop
    [do something] # Optional
for [second iterating variable] in [nested loop]: # Nested loop
    [do something]
```

The program first encounters the outer loop, executing its first iteration. This first iteration triggers the inner, nested loop, which then runs to completion. Then the program returns back to the top of the outer loop, completing the second iteration and again triggering the nested loop. Again, the nested loop runs to completion, and the program returns back to the top of the outer loop until the sequence is complete or a `break` or other statement disrupts the process.

Let's implement a nested `for` loop so we can take a closer look. In this example, the outer loop will iterate through a list of integers called `num_list`, and the inner loop will iterate through a list of strings called `alpha_list`.

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']
```

```
for number in num_list:
    print(number)
for letter in alpha_list:
```

```
print(letter)
```

When we run this program, we'll receive the following output:

Output

```
1
a
b
c
2
a
b
c
3
a
b
c
```

The output illustrates that the program completes the first iteration of the outer loop by printing 1, which then triggers completion of the inner loop, printing a, b, c consecutively. Once the inner loop has completed, the program returns to the top of the outer loop, prints 2, then again prints the inner loop in its entirety (a, b, c), etc.

Nested `for` loops can be useful for iterating through items within lists composed of lists. In a list composed of lists, if we employ just one `for` loop, the program will output each internal list as an item:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'], [0, 1, 2], [9.9, 8.8, 7.7]]
```

```
for list in list_of_lists:
    print(list)
```

Output

```
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
[9.9, 8.8, 7.7]
```

In order to access each individual item of the internal lists, we'll implement a nested `for` loop:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'], [0, 1, 2], [9.9, 8.8, 7.7]]
```

```
for list in list_of_lists:
```

```
for item in list:
```

```
    print(item)
```

Output

hammerhead

great white

dogfish

0

1

2

9.9

8.8

7.7

When we utilize a nested `for` loop we are able to iterate over the individual items contained in the lists.

## Chapter-6

# LOOPS EXAMPLES

### Examples

Program to take sum with the help of for and range

```
sum=0
for i in range(11):
    sum=sum+i
print sum
```

### Output

55

Now what we will do is find the sum of these no and sum as variable.

I can define another variable sum here and sum is going to be zero .now each time you go through the loop why don't you define sum to be the previous sum and what *i* is in this part of loop. Now print sum.

### **Explanation ::**

Think sum as buckets .

List - view them as referring to something.

Now reality is, it is now variable sum which is referring to numeric literal, the actual number zero sitting in the memory at someplace.

Now enter for loop, it says lets assign, lets iterative assign *i* to each of the element in the list generated by range of 11.

It will generate a list .

[0,1,2,3,4,5,6,7,8,9,10]

Now put *i* in second bucket and compare with list generated. Zero will go in *i* bucket and then print zero 'cause sum is zero. Next time he ask to range, are there any more soldier left . He said sure there are.

**That is way how it prints.**

### Indentation problem (Avoid it)

```
sum=0
for i in range(11):
    sum=sum+i
    print sum
```

### Output

```
0
1
3
6
10
15
21
28
36
45
55
```

This is called indentation problem. Now the last line is printing each and every step that for loop is iterating.

### Python Program to Add Two Numbers

```
# This program adds two numbers
```

```
num1 = 1.5
```

```
num2 = 6.3
```

```
# Add two numbers
```

```
sum = float(num1) + float(num2)
```

```
# Display the sum
```

```
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

### Output

```
The sum of 1.5 and 6.3 is 7.8
```

## Add Two Numbers Provided by The User

```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

### Output

```
Enter first number: 1.5
```

```
Enter second number: 6.3
```

```
The sum of 1.5 and 6.3 is 7.8
```

In this program, we asked user to enter two numbers and this program displays the sum of two numbers entered by user.

We use the built-in function `input()` to take the input. `input()` returns a string, so we convert it into number using the `float()` function.

## Python Program to Find the Square Root

```
# Note: change this value for a different result

num = 8

# uncomment to take the input from the user
```



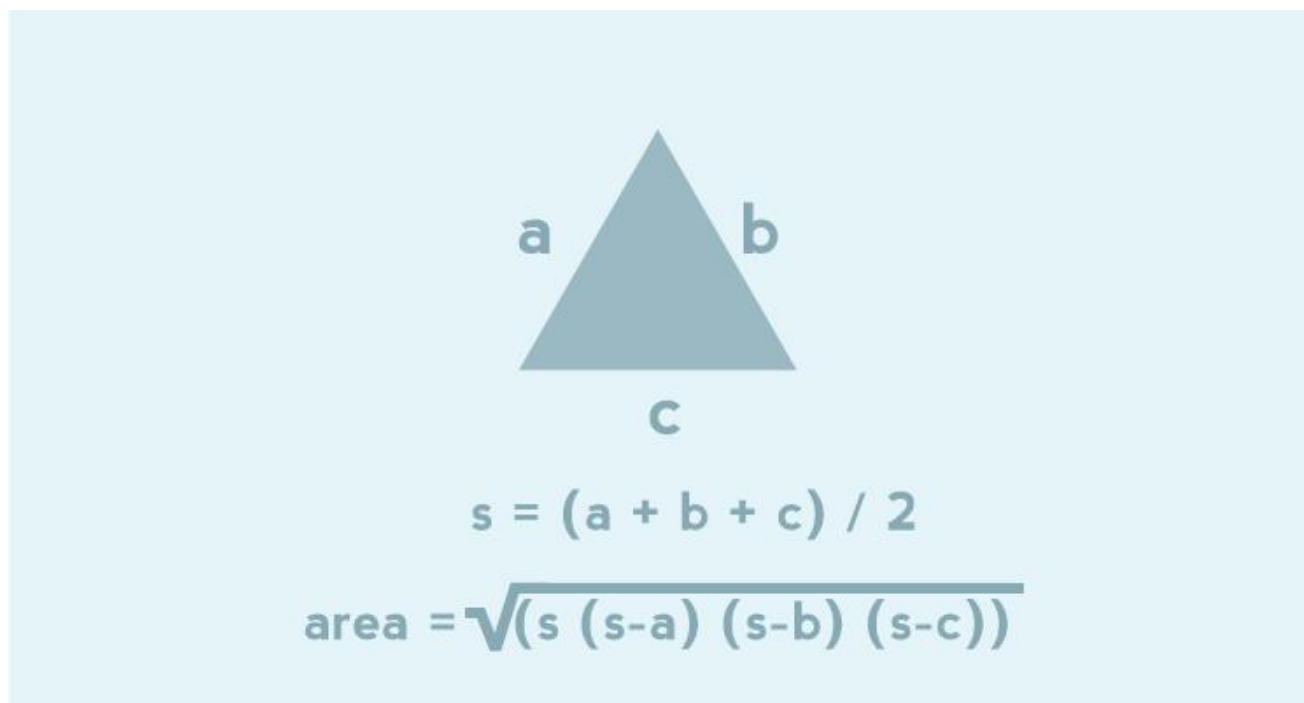
```
#num = float(input('Enter a number: '))  
  
num_sqrt = num ** 0.5  
  
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))
```

### Output

The square root of 8.000 is 2.828

## Python Program to Calculate the Area of a Triangle

In this program, you'll learn to calculate the area of a triangle and display it.



If  $a$ ,  $b$  and  $c$  are three sides of a triangle. Then,

```
s = (a+b+c)/2  
  
area = √(s(s-a)*(s-b)*(s-c))
```

```
a = 5  
  
b = 6  
  
c = 7  
  
# Uncomment below to take inputs from the user  
  
# a = float(input('Enter first side: '))  
  
# b = float(input('Enter second side: '))  
  
# c = float(input('Enter third side: '))  
  
# calculate the semi-perimeter  
  
s = (a + b + c) / 2  
  
# calculate the area  
  
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
```

```
print('The area of the triangle is %.2f' %area)
```

### Output

```
The area of the triangle is 14.70
```

## Python Program to Swap Two Variables

---

```
# To take input from the user

# x = input('Enter value of x: ')

# y = input('Enter value of y: ')

x = 5

y = 10

# create a temporary variable and swap the values

temp = x

x = y

y = temp

print('The value of x after swapping: {}'.format(x))

print('The value of y after swapping: {}'.format(y))
```

### Output

```
The value of x after swapping: 10
```

```
The value of y after swapping: 5
```

## Python Program to Generate a Random Number

```
# Program to generate a random number between 0 and 9

# import the random module

import random

print(random.randint(0,9))
```

### Output

5

Note that, we may get different output because this program generates random number in range 0 and 9. The syntax of this function is:

```
random.randint(a,b)
```

## Python program to check if a number is positive, negative or zero

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

Here, we have used the if...elif...else statement. We can do the same thing using nested if statements as follows.

### 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

A number is positive if it is greater than zero. We check this in the expression of `if`. If it is `False`, the number will either be zero or negative. This is also tested in subsequent expression.

## Python Program to Check if a Number is Odd or Even

```
# 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

In this program, we ask the user for the input and check if the number is odd or even.

## Python Program to Check Leap Year

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
```

### Python Program to Find the Largest Among Three Numbers

```
# Python program to find the largest number among the three input numbers
# change the values of num1, num2 and num3
# for a different result

num1 = 10
num2 = 14
num3 = 12
```

```
# uncomment following lines to take three numbers from user
#num1 = float(input("Enter first number: "))
#num2 = float(input("Enter second number: "))
#num3 = float(input("Enter third number: "))
if (num1 >= num2) and (num1 >= num3):
    largest = num1
elif (num2 >= num1) and (num2 >= num3):
    largest = num2
else:
    largest = num3
print("The largest number between",num1,",",num2,"and",num3,"is",largest)
```

### Output

```
The largest number between 10, 14 and 12 is 14.0
```

## Python Program to Print all Prime Numbers in an Interval

```
# Python program to display all the prime numbers within an interval
# change the values of lower and upper for a different result
lower = 900
upper = 1000
# uncomment the following lines to take input from the user
#lower = int(input("Enter lower range: "))
#upper = int(input("Enter upper range: "))
print("Prime numbers between",lower,"and",upper,"are:")
for num in range(lower,upper + 1):
```



```
# prime numbers are greater than 1
if num > 1:
    for i in range(2, num):
        if (num % i) == 0:
            break
        else:
            print(num)
```

### Output

Prime numbers between 900 and 1000 are:

907

911

919

929

937

941

947

953

967

971

977

983

991

997

## Python Program to Find the Factorial of a Number

---

The factorial of a number is the product of all the integers from 1 to that number.

For example, the factorial of 6 (denoted as 6!) is  $1*2*3*4*5*6 = 720$ . Factorial is not defined for negative numbers and the factorial of zero is one,  $0! = 1$ .

# Python program to find the factorial of a number provided by the user.

# change the value for a different result

num = 7

# uncomment to take input from the user

#num =int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero

if num < 0:

print("Sorry, factorial does not exist for negative numbers")

elif num == 0:

print("The factorial of 0 is 1")

else:

for i in range(1,num + 1):

```
factorial = factorial*i  
print("The factorial of",num,"is",factorial)
```

### Output

```
The factorial of 7 is 5040
```

## Python Program to Display the multiplication Table

```
''' Python program to find the  
multiplication table (from 1 to 10)'''  
  
num = 12  
  
# To take input from the user  
# num = int(input("Display multiplication table of? "))  
# use for loop to iterate 10 times  
for i in range(1, 11):  
    print(num,'x',i,'=',num*i)
```

### Output

```
12 x 1 = 12  
  
12 x 2 = 24  
  
12 x 3 = 36  
  
12 x 4 = 48  
  
12 x 5 = 60
```

$$12 \times 6 = 72$$

$$12 \times 7 = 84$$

$$12 \times 8 = 96$$

$$12 \times 9 = 108$$

$$12 \times 10 = 120$$

## Python Program to Convert Decimal to Binary, Octal and Hexadecimal

Decimal system is the most widely used number system. But computer only understands binary. Binary, octal and hexadecimal number systems are closely related and we may require to convert decimal into these systems. Decimal system is base 10 (ten symbols, 0-9, are used to represent a number) and similarly, binary is base 2, octal is base 8 and hexadecimal is base 16.

A number with the prefix '0b' is considered binary, '0o' is considered octal and '0x' as hexadecimal. For example:

$$60 = 0b111100 = 0o74 = 0x3c$$

```
# Python program to convert decimal number into binary, octal and  
hexadecimal number system
```

```
# Change this line for a different result
```

```
dec = 344
```

```
print("The decimal value of",dec,"is:")
```

```
print(bin(dec),"in binary.")
```

```
print(oct(dec),"in octal.")
```

```
print(hex(dec),"in hexadecimal.")
```

## Output

The decimal value of 344 is:

0b101011000 in binary.

0o530 in octal.

0x158 in hexadecimal.

## Python Program to Find ASCII Value of Character

---

```
# Program to find the ASCII value of the given character
# Change this value for a different result
c = 'p'
# Uncomment to take character from user
#c = input("Enter a character: ")
print("The ASCII value of '" + c + "' is",ord(c))
```

## Output 1

The ASCII value of 'p' is 112

## Python Program to Find Factors of Number

---

```
x=10
y=[]
for i in range(1, x + 1):
    if x % i == 0:
        y.append(i)
print(y)
```

## Output

```
[1, 2, 5, 10]
```

## Python Program to Check Whether a String is Palindrome or Not

```
# Program to check if a string
# is palindrome or not
# change this value for a different output
my_str = 'aIbohPhoBiA'
# make it suitable for caseless comparison
my_str = my_str.casefold()
# reverse the string
rev_str = reversed(my_str)
# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
    print("It is palindrome")
else:
    print("It is not palindrome")
```

## Output

```
It is palindrome
```

## Python Program to Remove Punctuations From a String

```
# define punctuation
punctuations = '''!()-[]{};:'"\.,<>./?@#$$%^&*~'''
```

```
my_str = "Hello!!!, he said ---and went."  
# To take input from the user  
# my_str = input("Enter a string: ")  
# remove punctuation from the string  
no_punct = ""  
for char in my_str:  
    if char not in punctuations:  
        no_punct = no_punct + char  
# display the unpunctuated string  
print(no_punct)
```

### Output

```
Hello he said and went
```

## Python Program to Sort Words in Alphabetic Order

```
# Program to sort alphabetically the words form a string provided by the  
user  
  
# change this value for a different result  
my_str = "i am a boy"  
  
# uncomment to take input from the user  
#my_str = input("Enter a string: ")  
  
# breakdown the string into a list of words  
words = my_str.split()  
  
# sort the list  
words.sort()
```

```
# display the sorted words
print("The sorted words are:")
for word in words:
    print(word)
```

### Output

```
The sorted words are:
```

```
a
```

```
am
```

```
boy
```

```
i
```

### Python Program to Count the Number of Each Vowel

```
vowels = 'aeiou'

# change this value for a different result
ip_str = 'Hello, have you tried our tutorial section yet?'

# uncomment to take input from the user
#ip_str = input("Enter a string: ")

# make it suitable for caseless comparisons
ip_str = ip_str.casefold()

# make a dictionary with each vowel a key and value 0
count = {}.fromkeys(vowels,0)
```



```
# count the vowels  
for char in ip_str:  
    if char in count:  
        count[char] += 1  
print(count)
```

### Output

```
{'o': 5, 'i': 3, 'a': 2, 'e': 5, 'u': 3}
```

# Chapter-7

## PATTERNS USING NESTED LOOPS

### Programs for printing patterns

Patterns can be printed in python using simple for loops. **First outer loop** is used to handle **number of rows** and **Inner nested loop** is used to handle the **number of columns**. Manipulating the print statements, different number patterns, alphabet patterns or star patterns can be printed.

### Simple pyramid pattern

# Python 3.x code to demonstrate star pattern

```
# outer loop to handle number of rows
# n in this case
n=5
fori inrange(0, n):
    # inner loop to handle number of columns
    # values changing acc. to outer loop
    forj inrange(0, i+1):

        # printing stars
        print("* ",end="")

    # ending line after each row
    print("\r")
```

Output:

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

# Python 3.x code to demonstrate star pattern

# Function to demonstrate printing pattern

```
# number of spaces
n=5
k =2*n -2

# outer loop to handle number of rows
fori inrange(0, n):
```

```
# inner loop to handle number spaces
# values changing acc. to requirement
forj inrange(0, k):
    print(end=" ")

# decrementing k after each loop
k =k -2

# inner loop to handle number of columns
# values changing acc. to outer loop
forj inrange(0, i+1):

    # printing stars
    print("* ", end="")

# ending line after each row
print("\r")
```

Output:

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

## Printing Triangle

```
# Python 3.x code to demonstrate star pattern
# Function to demonstrate printing pattern triangle
# number of spaces
n=5
k =2*n -2
# outer loop to handle number of rows
fori inrange(0, n):

    # inner loop to handle number spaces
    # values changing acc. to requirement
    forj inrange(0, k):
        print(end=" ")

    # decrementing k after each loop
    k =k -1

    # inner loop to handle number of columns
    # values changing acc. to outer loop

    forj inrange(0, i+1):
```

```
# printing stars
print("* ", end="")

# ending line after each row
print("\r")
```

Output:

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

## Number Pattern

```
# Python 3.x code to demonstrate star pattern
# Function to demonstrate printing pattern of numbers

# initialising starting number
n=5
num =1

# outer loop to handle number of rows
fori inrange(0, n):

    # re assigning num
    num =1
    # inner loop to handle number of columns
    # values changing acc. to outer loop
    forj inrange(0, i+1):

        # printing number
        print(num, end=" ")

        # incrementing number at each column
        num =num +1

    # ending line after each row
    print("\r")
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

## Numbers without re assigning

```
# Python 3.x code to demonstrate star pattern
# initializing starting number
n=5
num =1

# outer loop to handle number of rows
fori inrange(0, n):

    # not re assigning num
    # num = 1

    # inner loop to handle number of columns
    # values changing acc. to outer loop
    forj inrange(0, i+1):

        # printing number
        print(num, end=" ")

        # incrementing number at each column
        num =num +1

    # ending line after each row
    print("\r")
```

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

## Character Pattern

```
# Python 3.x code to demonstrate star pattern
# initializing value corresponding to 'A'
# ASCII value
n=5
num =65

# outer loop to handle number of rows
# 5 in this case
fori inrange(0, n):

    # inner loop to handle number of columns
    # values changing acc. to outer loop
    forj inrange(0, i+1):

        # explicitly converting to char
        ch =chr(num)

        # printing char value
        print(ch, end=" ")

    # incrementing number
    num =num +1

    # ending line after each row
    print("\r")
```

Output:

```
A
B B
C C C
D D D D
E E E E E
```

## Continuous Character pattern

```
# Python code 3.x to demonstrate star pattern
# initializing value corresponding to 'A'
# ASCII value
n=5
num =65
# outer loop to handle number of rows
- fori inrange(0, n):

    # inner loop to handle number of columns
    # values changing acc. to outer loop
    forj inrange(0, i+1):
```

```
# explicitly converting to char
ch =chr(num)

# printing char value
print(ch, end=" ")

# incrementing at each column
num =num +1


# ending line after each row
print("\r")
```

- Output:

```
A
B C
D E F
G H I J
K L M N O
```

# Chapter-8

## FUNCTIONS

### FUNCTIONS

Way to combine many actions into a single process, for the situations where you will be using it over and over again.

Functions would be collections of actions. Ex- cleaning of kitchen which includes wash dishes, sweep moping.

Functions are really awesome because they allow you to do complex things without going through each of the steps again and again. It helps you to use that code over and over again. For ex if you want to do something in your code few times then you can just put it in your function and call that function where ever and whenever you want. Just remember not to use legal names in it. Now comes the part which is called body of function. It basically just contains everything you want with that function to do. The idea of functions is to assign a self ode and variables known as parameters through a single bit of a text. So to begin a function, you define a function using keyword **def**. This tells python that we are using function. After that we type whatever name we wanna refer to your function. Just be little carefull with the name of your function. Choose something unique that doesn't conflict with anything else.

For ex if we type `def time()`.

It 'll conflict with time module. So you don't wanna use time.

### 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`.

## RETURN:

The `print()` function writes, i.e., "prints", a string in the console. The `return` statement causes your function to exit and hand back a value to its caller. The point of functions in general is to take in inputs and return something. The `return` statement is used when a function is ready to return a value to its caller.

Return is a way to return something (or data) back from a function. It's a form of output.

The `return` keyword is used to return values from a function. A function may or may not return a value. If a function does not have a `return` keyword, it will send a `None` value.

Arguments are used for input into functions and `return` is used for output

NOW , how do we push stuff in function ?

- That is with arguments. Arguments are way of passing information into a function within the parenthesis. It's a form of input. Arguments are a way of inputting into a function.
- Ex- The same way you put gas into a car .Putting gas is input and due to that car runs is output which is `return`.

Now you will understand the use of functions with these 3 examples

### Example 1

```
def demo():  
    return "Hello"  
    z=2+2  
    return z  
print demo()
```

Output:

```
Hello
```

```
4
```

## Example 2

```
def demo(a,b):  
    return a+b  
print demo(2,3)  
print demo(3,3)  
  
print demo(2,4)  
print demo(2,7)
```

Output:

```
5  
6  
6  
9
```

## Example 3

```
def demo(a,b):  
    return a+b  
    return a-b  
    return a*b  
    return a/b  
print demo(10,2)
```

Output:

```
12  
8  
20  
5
```

## 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 –

```
• # 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 –

- 
- `# 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 –

- `Mystring`

- The following example gives more clear picture. Note that the order of parameters does not matter.

- `# 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`
- `Age50`

## 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 –

```
• # 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
• Age50
• Name:miki
• Age35
```

## 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 comp(*var):
•     Print var
•     comp(1,2,5,7,8)
```

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

```
• Output is: 1,2,5,7,8
```

**We can also use the `**` construct in our functions. In such a case, the function will accept a dictionary. The dictionary has arbitrary length. We can then normally parse the dictionary, as usual.**

**can then normally parse the dictionary, as usual.**

```
#!/usr/bin/python

def display(**details):

    for i in details:
        print "%s: %s" % (i, details[i])

display(name="Lary", age=43, sex="M")
```

This example demonstrates such a case. We can provide arbitrary number of key-value arguments. The function will handle them all.

```
age: 43
name: Lary
sex: M
```

## The Anonymous Functions

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".

```
>>> def f (x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>>
>>> print g(8)
64
```

## 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

### Example 1: The difference between global and local variables

Global variables are accessible inside and outside of functions. Local variables are only accessible inside the function. In the example below, the function can access both the global and the local variable. However, trying to access the local variable outside the function produces an error.

```
global_var='foo'
def ex1():
    local_var='bar'
    print global_var
    print local_var

ex1()
print global_var
print local_var # this gives an error
foo
bar
foo
Traceback (most recent call last):
  File "nested_scope.py", line 12, in
    print local_var # this gives an error
NameError: name 'local_var' is not defined
```

### Example 2: How \*not\* to set a global variable

\*Setting\* a global variable from within a function is not as simple. If I set a variable in a function with the same name as a global variable, I am actually creating a new local variable. In the example below, var remains 'foo' even after the function is called.

```
var='foo'
```

```
def ex2():  
    var='bar'  
    print 'inside the function var is ',var  
  
ex2()  
print 'outside the function var is ',var  
inside the function var is  bar  
outside the function var is  foo
```

### Example 3: How to set a global variable

To set the global variable inside a function, I need to use the `global` statement. This declares the inner variable to have module scope. Now `var` remains 'bar' after the function is called.

```
var='foo'  
def ex3():  
    global var  
    var='bar'  
    print 'inside the function var is ',var  
  
ex3()  
print 'outside the function var is ',var  
inside the function var is  bar  
outside the function var is  bar
```



# Chapter-9

## FUNCTION EXAMPLES

### Python functions Problems

1. Write a Python function to find the Max of three numbers.

2. Write a Python function to sum all the numbers in a list.

*Sample List : (8, 2, 3, 0, 7)*

*Expected Output : 20*

3. Write a Python function to multiply all the numbers in a list.

*Sample List : (8, 2, 3, -1, 7)*

*Expected Output : -336*

4. Write a Python program to reverse a string.

*Sample String : "1234abcd"*

*Expected Output : "dcba4321"*

5. Write a Python function to calculate the factorial of a number (non-negative integer). The function accept the number as a argument.

6. Write a Python function to check whether a number is in a given range.

7. Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

*Sample String : 'The quick Brow Fox'*

*Expected Output :*

No. of Upper case characters : 3

No. of Lower case Characters : 12

8. Write a Python function that takes a list and returns a new list with unique elements of the first list.

*Sample List : [1,2,3,3,3,3,4,5]*

*Unique List : [1, 2, 3, 4, 5]*

9. Write a Python function that takes a number as a parameter and check the number is prime or not.

Note : A prime number (or a prime) is a natural number greater than 1 and that has no positive divisors other than 1 and itself.

**10.** Write a Python program to print the even numbers from a given list. *Sample List :*  
[1, 2, 3, 4, 5, 6, 7, 8, 9]  
*Expected Result :* [2, 4, 6, 8]

**12.** Write a Python function that checks whether a passed string is palindrome or not.  
Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

**15.** Write a Python program that accepts a hyphen separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.  
*Sample Items :* green-red-yellow-black-white  
*Expected Result :* black-green-red-white-yellow

**16.** Write a Python function to create and print a list where the values are square of numbers between 1 and 30 (both included).

## Solutions

### ANSWERS

1.

```
view plain copy to clipboard print ?
01. def max_of_two( x, y ):
02.     if x > y:
03.         return x
04.     return y
05. def max_of_three( x, y, z ):
06.     return max_of_two( x, max_of_two( y, z ) )
07. print(max_of_three(3, 6, -5))
```

2.

```
view plain copy to clipboard print ?
01. def sum(numbers):
02.     total = 0
03.     for x in numbers:
04.         total += x
05.     return total
06. print(sum((8, 2, 3, 0, 7)))
```

3.

```
view plain copy to clipboard print ?
01. def multiply(numbers):
02.     total = 1
03.     for x in numbers:
04.         total *= x
05.     return total
06. print(multiply((8, 2, 3, -1, 7)))
```

4.

```
view plain copy to clipboard print ?
01. def string_reverse(str1):
02.
03.     rstr1 = ''
04.     index = len(str1)
05.     while index > 0:
06.         rstr1 += str1[ index - 1 ]
07.         index = index - 1
08.     return rstr1
09. print(string_reverse('1234abcd'))
```

5.

```
view plain copy to clipboard print ?
01. def factorial(n):
02.     if n == 0:
03.         return 1
04.     else:
05.         return n * factorial(n-1)
06. n=int(input("Input a number to compute the factiorial : "))
07. print(factorial(n))
```

6.

```
view plain copy to clipboard print ?
01. def test_range(n):
02.     if n in range(3,9):
03.         print( " %s is in the range"%str(n))
04.     else :
05.         print("The number is outside the given range.")
06. test_range(5)
```

7.

```
view plain copy to clipboard print ?
01. def string_test(s):
02.     d={"UPPER_CASE":0, "LOWER_CASE":0}
03.     for c in s:
04.         if c.isupper():
05.             d["UPPER_CASE"]+=1
06.         elif c.islower():
07.             d["LOWER_CASE"]+=1
08.         else:
09.             pass
10.     print ("Original String : ", s)
11.     print ("No. of Upper case characters : ", d["UPPER_CASE"])
12.     print ("No. of Lower case Characters : ", d["LOWER_CASE"])
13.
14. string_test('The quick Brow Fox')
```

8.

```
view plain copy to clipboard print ?
01. def unique_list(l):
02.     x = []
03.     for a in l:
04.         if a not in x:
05.             x.append(a)
06.     return x
07.
08. print(unique_list([1,2,3,3,3,3,4,5]))
```

9.

```
view plain copy to clipboard print ?
01. def test_prime(n):
02.     if (n==1):
03.         return False
04.     elif (n==2):
05.         return True;
06.     else:
07.         for x in range(2,n):
08.             if(n % x==0):
09.                 return False
10.         return True
11. print(test_prime(9))
```

10.

```
view plain copy to clipboard print ?
01. def is_even_num(l):
02.     enum = []
03.     for n in l:
04.         if n % 2 == 0:
05.             enum.append(n)
06.     return enum
07. print(is_even_num([1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

11.

```
view plain copy to clipboard print ?
01. def isPalindrome(string):
02.     left_pos = 0
03.     right_pos = len(string) - 1
04.
05.     while right_pos >= left_pos:
06.         if not string[left_pos] == string[right_pos]:
07.             return False
08.         left_pos += 1
09.         right_pos -= 1
10.     return True
11. print(isPalindrome('aza'))
```

12.

```
view plain copy to clipboard print ?
01. items=[n for n in input().split('-')]
02. items.sort()
03. print('-'.join(items))
```

13.

```
view plain copy to clipboard print ?
01. def printValues():
02.     l = list()
03.     for i in range(1,21):
04.         l.append(i**2)
05.     print(l)
06.
07. printValues()
```

# Chapter-10

## MODULES & PACKAGES

### Modules and packages

Any Python file is a [module](#), its name being the file's base name without the .py extension. A [package](#) is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional `__init__.py` file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts.

### How to Create a Python Module

We will begin by creating a super simple module. This module will provide us with basic arithmetic and no error handling. Here's our first example:

```
#-----  
  
def add(x, y):  
    """  
  
    return x + y  
  
    """  
  
#-----  
  
def division(x, y):  
    """  
  
    return x / y  
  
    """  
  
#-----  
  
def multiply(x, y):  
    """  
  
    """
```

```
return x * y
```

```
#-----
```

```
def subtract(x, y):
```

```
    """
```

```
    return x - y
```

Let's call it arithmetic.py. Now what can you do with a module anyway? You can import it and use any of the defined modules.

First we'll write a little script that imports our module and runs the functions in it:

```
import arithmetic
```

```
print(arithmetic.add(5, 8))
```

```
print(arithmetic.subtract(10, 5))
```

```
print(arithmetic.division(10, 5))
```

```
print(arithmetic.multiply(12, 6))
```

OUTPUT

```
13
```

```
5
```

```
2
```

```
72
```

## How to Create a Python Package

The main difference between a module and a package is that a package is a collection of modules AND it has an `__init__.py` file. Depending on the complexity of the package, it may have more than one `__init__.py`. Let's take a look at a simple folder structure to make this more obvious, then we'll create some simple code to follow that structure.

```
myMath/  
  
__init__.py  
  
adv/  
  
__init__.py  
  
    sqrt.py  
  
    fib.py  
  
    add.py  
  
    subtract.py  
  
    multiply.py  
  
    divide.py
```

Now we just need to replicate this structure in our own package. Let's give that a whirl! Create each of these files in a folder tree like the above. For the add, subtract, multiply and divide files, you can use the functions we created in the earlier example. For the other two, we'll use the following code.

For the fibonacci sequence, we'll use this simple code from StackOverflow:

```
# fib.py  
  
from math import sqrt  
  
  
#-----
```



```
def fibonacci(n):  
  
    """  
    http://stackoverflow.com/questions/494594/how-to-write-the-fibonacci-sequence-in-python  
    """  
  
    return ((1+sqrt(5))**n - (1-sqrt(5))**n) / (2**n*sqrt(5))
```

For the `sqrt.py` file, we'll use this code:

```
# sqrt.py  
  
import math  
  
#-----  
  
def squareroot(n):  
  
    """  
    """  
  
    return math.sqrt(n)
```

You can leave both `__init__.py` files blank, but then you'll have to write code like `mymath.add.add(x,y)` which kind of sucks, so we'll add the following code to the outer `__init__.py` to make using our package easier.

```
# outer __init__.py  
  
from add import add  
  
from divide import division  
  
from multiply import multiply  
  
from subtract import subtract  
  
from adv.fib import fibonacci
```

```
fromadv.sqrtimportsquareroot
```

Now we should be able to use our module once we have it on our Python path. You can copy the folder into your Python's site-packages folder to do this. On Windows it's in the following general location: C:\Python26\Lib\site-packages. Alternatively, you can edit the path on the fly in your test code. Let's see how that's done:

```
importsys

sys.path.append('C:\Users\mdriscoll\Documents')

importmymath

printmymath.add(4,5)

printmymath.division(4, 2)

printmymath.multiply(10, 5)

printmymath.fibonacci(8)

printmymath.squareroot(48)
```

Note that my path does NOT include the mymath folder. You want to append the parent folder that holds your new module, NOT the module folder itself.

# Chapter-11

## FILE HANDLING (TXT AND CSV)

### Reading and Writing Files in Python

When you're working with Python, you don't need to import a library in order to read and write files. It's handled natively in the language.

The first thing you'll need to do is use Python's built-in *open* function to get a *file object*.

The *open* function opens a file. It's simple.

When you use the *open* function, it returns something called a *file object*. *File objects* contain methods and attributes that can be used to collect information about the file you opened. They can also be used to manipulate said file.

For example, the *mode* attribute of a *file object* tells you which mode a file was opened in. And the *name* attribute tells you the name of the file that the *file object* has opened.

You must understand that a *file* and *file object* are two wholly separate – yet related – things.

### File Types

In Python, a file is categorized as either text or binary, and the difference between the two file types is important.

Text files are structured as a sequence of lines, where each line includes a sequence of characters. This is what you know as code or syntax.

Each line is terminated with a special character, called the EOL or **End of Line** character. There are several types, but the most common is the comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

A backslash character can also be used, and it tells the interpreter that the next character – following the slash – should be treated as a new line. This character is useful when you don't want to start a new line in the text itself but in the code.

## Open ( ) Function

In order to open a file for writing or use in Python, you must rely on the built-in *open* () function.

As explained above, *open* ( ) will return a file object, so it is most commonly used with two arguments.

An argument is nothing more than a value that has been provided to a function, which is relayed when you call it. So, for instance, if we declare the name of a file as “Test File,” that name would be considered an argument.

The syntax to open a file object in Python is:

`file_object = open("filename", "mode")` where `file_object` is the variable to add the file object.

The second argument you see – *mode* – tells the interpreter and developer which way the file will be used.

## Mode

Including a mode argument is optional because a default value of ‘*r*’ will be assumed if it is omitted. The ‘*r*’ value stands for read mode, which is just one of many.

The modes are:

- ‘*r*’ – Read mode which is used when the file is only being read
- ‘*w*’ – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)
- ‘*a*’ – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end
- ‘*r+*’ – Special read and write mode, which is used to handle both actions when working with a file

So, let’s take a look at a quick example.

`F = open("workfile", "w")`

Print f

This snippet opens the file named “workfile” in writing mode so that we can make changes to it. The current information stored within the file is also displayed – or printed – for us to view.

Once this has been done, you can move on to call the objects functions. The two most common functions are read and write.

## Create a text file

To get more familiar with text files in Python, let's create our own and do some additional exercises.

Using a simple text editor, let's create a file. You can name it anything you like, and it's better to use something you'll identify with.

However, we are going to call it "testfile.txt".

Just create the file and leave it blank.

To manipulate the file, write the following in your Python environment (you can copy and paste if you'd like):

```
file = open("testfile.txt", "w")  
  
file.write("Hello World")  
file.write("This is our new text file")  
file.write("and this is another line.")  
file.write("Why? Because we can.")  
  
file.close()
```

Naturally, if you open the text file – or look at it – using Python you will see only the text we told the interpreter to add.

```
$ cat testfile.txt  
Hello World  
This is our new text file  
and this is another line.
```

Why? Because we can.

## Reading a Text File in Python

There are actually a number of ways to read a text file in Python, not just one.

If you need to extract a string that contains all characters in the file, you can use the following method:

```
file.read()
```

The full code to work with this method will look something like this:

```
file = open("testfile.txt", "r")  
printfile.read()
```

The output of that command will display all the text inside the file, the same text we told the interpreter to add earlier. There's no need to write it all out again, but if you must know, everything will be shown except for the "\$ cat testfile.txt" line.

Another way to read a file is to call a certain number of characters.

For example, with the following code the interpreter will read the first five characters of stored data and return it as a string:

```
file = open("testfile.txt", "r")  
  
printfile.read(5)
```

Notice how we're using the same *file.read()* method, only this time we specify the number of characters to process?

The output for this will look like:

```
Hello
```

If you want to read a file line by line – as opposed to pulling the content of the entire file at once – then you use the `readline()` function.

Why would you use something like this?

Let's say you only want to see the first line of the file – or the third. You would execute the `readline()` function as many times as possible to get the data you were looking for.

Each time you run the method, it will return a string of characters that contains a single line of information from the file.

```
file = open("testfile.txt", "r")  
printfile.readline()
```

This would return the first line of the file, like so:

```
Hello World
```

If we wanted to return only the third line in the file, we would use this:

```
file = open("testfile.txt", "r")  
printfile.readline(3)
```

But what if we wanted to return every line in the file, properly separated? You would use the same function, only in a new form. This is called the `file.readlines()` function.

```
file = open("testfile.txt", "r")  
printfile.readlines()
```

The output you would get from this is:

```
['Hello World', 'This is our new text file', 'and this is another line.', 'Why? Because we can.']
```

Notice how each line is separated accordingly? Note that this is not the ideal way to show users the content in a file. But it's great when you want to collect information quickly for personal use during development or recall.

## Looping over a file object

When you want to read – or return – all the lines from a file in a more memory efficient, and fast manner, you can use the loop over method. The advantage to using this method is that the related code is both simple and easy to read.

```
file = open("testfile.txt", "r")  
for line in file:  
    print line,
```

This will return:

```
Hello World  
This is our new text file  
and this is another line.  
Why? Because we can.
```

## Closing a File

When you're done working, you can use the `fh.close()` command to end things. What this does is close the file completely, terminating resources in use, in turn freeing them up for the system to deploy elsewhere.

It's important to understand that when you use the `fh.close()` method, any further attempts to use the file object will fail.



## File Handling in the Real World

To help you better understand some of the methods discussed here, we're going to offer a few examples of them being used in the real world. Feel free to copy the code and try it out for yourself in a Python interpreter (make sure you have any named files created and accessible first).

Opening a text file:

```
fh = open("hello.txt", "r")
```

Reading a text file:

```
Fh= open("hello.txt", "r")  
printfh.read()
```

To read a text file one line at a time:

```
fh = open("hello.text", "r")  
printfh.readline()
```

To read a list of lines in a text file:

```
fh = open("hello.txt", "r")  
printfh.readlines()
```

To write new content or text to a file:

```
fh=open("hello.txt", "w")  
  
fh.write("Put the text you want to add here")  
fh.write("and more lines if need be.")  
  
fh.close()
```

You can also use this to write multiple lines to a file at once:

```
fh = open("hello.txt", "w")  
lines_of_text = ["One line of text here", "and another line here", "and yet another  
here", "and so on and so forth"]  
fh.writelines(lines_of_text)  
fh.close()
```

To append a file:

```
fh = open("hello.txt", "a")  
fh.write("We Meet Again World")  
fh.close
```

To close a file completely when you are done:

```
fh = open("hello.txt", "r")  
printfh.read()  
fh.close()
```

## 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()
```

## Using the CSV module in Python

In this we learn how to read CSV data in from a file and then use it in Python. For this, we use the csv module. CSV literally stands for comma separated variable, where the comma is what is known as a "delimiter." While you can also just simply use Python's `split()` function, to separate lines and data within each line, the CSV module can also be used to make things easy.

Example CSV file data:

```
1/2/2014,5,8,red
1/3/2014,5,2,green
1/4/2014,9,1,blue
```

Next, let's cover the reading of CSV files into memory:

```
import csv

with open('example.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    for row in readCSV:
        print(row)
        print(row[0])
        print(row[0], row[1], row[2],)
```

Above, we've shown how to open a CSV file and read each row, as well as reference specific data on each row.

Next, we will show how to pull out specific data from the spreadsheet and save it to a list variable:

```
import csv

with open('example.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    dates = []
    colors = []
    for row in readCSV:
        color = row[3]
        date = row[0]

        dates.append(date)
        colors.append(color)

    print(dates)
    print(colors)
```

Once we have this data, what can we do with it? Maybe we are curious about what color something was on a specific date.

```
import csv

with open('example.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    dates = []
    colors = []
    for row in readCSV:
        color = row[3]
        date = row[0]

        dates.append(date)
```

```
colors.append(color)

print(dates)
print(colors)

# now, remember our lists?

whatColor= input('What color do you wish to know the date of?:')
coldex=colors.index(whatColor)
theDate= dates[coldex]
print('The date of',whatColor,'is:',theDate)
```

# Chapter-12

## EXCEPTIONAL HANDLING

### What is an Exception?

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Lets take an example

```
for i in range(-5,5):  
    print (100/i)
```

### OUTPUT

```
-20  
-25  
-34  
-50  
-100  
Traceback (most recent call last):  
  
  File "<ipython-input-2-3466829ec8ae>", line 1, in <module>  
    runfile('C:/Users/DracOniS/OneDrive/Python Scripts/Exceptional handling.py',  
wdir='C:/Users/DracOniS/OneDrive/Python Scripts')  
  
  File "C:\Users\DracOniS\Anaconda\lib\site-  
packages\spyderlib\widgets\externalshell\sitecustomize.py", line 580, in runfile  
    execfile(filename, namespace)  
  
  File "C:/Users/DracOniS/OneDrive/Python Scripts/Exceptional handling.py", line 17, in <module>  
    print(100/i)  
  
ZeroDivisionError: integer division or modulo by zero
```

So, it stopped at zero and division of positive numbers was not executed. So how to bypass that. This is when exceptional handling comes to play.



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

- **Exception Handling:** This would be covered in this session.
- **Assertions:** This would be covered in Assertions in Python .

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.

AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError EnvironmentError	<p>Raised when trying to access a local variable in a function or method but no value has been assigned to it.</p> <p>Base class for all exceptions that occur outside the Python environment.</p>
IOError IOError	<p>Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.</p> <p>Raised for operating system-related errors.</p>

SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

## Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

## The `assert` Statement

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

The syntax for `assert` is –

```
assertExpression[,Arguments]
```

## Example

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

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

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

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

## Handling an exception

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

## Syntax

Here is simple syntax of *try....except...else* blocks –

```
try:
    Youdo your operations here;
    .....
exceptExceptionI:
    If there isExceptionI,then execute this block.
exceptExceptionII:
    If there isExceptionII,then execute this block.
    .....
else:
    If there isno exception then execute this block.
```

Here are few important points about the above-mentioned syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

## Example

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

```
#!/usr/bin/python

try:
    fh = open("testfile","w")
    fh.write("This is my test file for exception handling!!")
```

```
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
    fh.close()
```

This produces the following result –

```
Written content in the file successfully
```

## Example

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

```
#!/usr/bin/python

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

This produces the following result –

```
Error: can't find file or read data
```

## Example

This example tries to stop the error that comes when number is divided by zero.

```
try:
    print 1/0
except ZeroDivisionError:
    print "You can't divide by zero, you're silly."
```

This produces the following result –

```
You can't divide by zero, you're silly
```

## Example

```
# import module sys to get the type of exception
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
print("The reciprocal of",entry,"is",r)
```

## Output

```
The entry is a

Oops! <class 'ValueError'> occured.

Next entry.

The entry is 0
```

```
Oops! <class 'ZeroDivisionError' > occurred.
```

```
Next entry.
```

```
The entry is 2
```

```
The reciprocal of 2 is 0.5
```

## Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the `except` clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an `except` clause will catch.

A `try` clause can have any number of `except` clause to handle them differently but only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an `except` clause. Here is an example pseudo code.

```
try:
    # do something
    pass

except ValueError:
    # handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass

except:
```



```
# handle all other exceptions  
pass
```

## Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword `raise`.

We can also optionally pass in value to the exception to clarify why that exception was raised.

```
>>>raiseKeyboardInterrupt  
Traceback(most recent call last):  
...  
KeyboardInterrupt  
  
>>>raiseMemoryError("This is an argument")  
Traceback(most recent call last):  
...  
MemoryError:This is an argument  
  
>>>try:  
...     a =int(input("Enter a positive integer: "))  
...     if a <=0:  
...         raiseValueError("That is not a positive number!")  
... exceptValueErroras ve:  
...     print(ve)  
...  
Enter a positive integer:-2  
That is not a positive number!
```

## try...finally

The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

Here is an example of [file operations](#) to illustrate this.

```
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

This type of construct makes sure the file is closed even if an exception occurs.

## Chapter-13

# CLASSES & OBJECTS

## CLASSES

- Classes in python are how you make objects.
- The classes themselves are the blueprint for how an object is created.
- For Ex-
- Class Duck:
- `def quack(self):`
- `Print("Quaaaaaaaaaack")`
- `def walk(self):`
- `Print("walks like a duck")`

Here is the class called duck which implements couple of method called quack and walk. So when you'll create the duck objects it will

**"Quaaaaaaaaaack"** like a duck and will **"walk like a duck"**.

- An object is a instance of class. Ex –
- `Donald= Duck()`
- `Donald.quack()`
- `Donald.walk()`

When you create a **object**, it's a separate thing, its separately encapsulated, it has all of its own attribute. So if you make several different object from a given class each of those different instance of class, but they all are separately encapsulated and have their own data space, code space. An **object** is instance of class that means when you create an object that object is built from the **blueprint** of the class but it has its own object, its own encapsulation and so if you create several different **object** from the same **class** they are separately encapsulated.

In this example we have created a object called Donald from the duck class. Now we can say that Donald will quaaaaaaaaaack like a duck and walk like a duck. And therefore is a duck.

## WORKING –

Class is the keyword that introduces the definition of class. "Duck" here is the name of the class and now there is the colon because of colon, things will start

indenting and methods will be inside it. Now we have two things inside the class. We 've a function called quack and now this function have a self as its 1<sup>st</sup>

argument and now we have “walk” the 2<sup>nd</sup> method. Now if we go down we see a object called Donald. We created the object called Donald (which is instance of Duck) by assigning it to Duck and Duck is the class definition. Now Donald is an object of class duck .

**So, [Donald.quack]** – When I called the quack method we used this “.” Operator and this “.” Operator is attribute de reference operator which is used to reference an attribute of the object. In this case attribute is the method, so it’s called with the parenthesis. This means it’s gonna look inside the object Donald for an attribute called “quack” and because it has parenthesis on it , it’s going to call it as a object method and this will call **quack** method and print “quaaaaaaaaaack”. Donald here is the instance of Duck and now you can call methods inside the Donald and So you can call the methods inside the Donald object of class duck using this “.” Operator.

The 1<sup>st</sup> argument to these function is “self “ and that is reference to the object not the class but the object . In other words when quack(from object) gets called on the Donald object then the Donald object gets passed to the “quack” method. So quack has a way of referring to object attributes. We see that nothing is passed inside these parenthesis in object . It’s as if Donald was passed inside those parenthesis but you don’t actually put that there that happens automatically by virtue of “.” Operator. When quack gets called as a method of Donald then Donald gets passes magically inside those parenthesis.

**Donald.quack(donald)** # magically, we don’t put it there.

**Donald.walk()**

So self is 1<sup>st</sup> argument to the method.

## **CONSTRUCTOR(\_\_init\_\_)**

Now there is special type of method that we ‘ll going to talk about which is called **constructor**. And this gets called everytime you create a objects based on this class. The constructor in python is created by naming a method by “\_\_init\_\_” and this creates a constructor method .

So when we run program , the 1<sup>st</sup> thing we do is we create a Donald object and that’s where constructor gets called . Infact one of the most common purposes for a constructor is to initialize some data . If I put in some number 47 here.

Donald=Duck(47)

And then I pass value in the constructor like `__init__(self,value):`

now we 'll save that value instead of printing constructor here by `self.v=value`.

What it does is that it creates a local variable that is an attribute of Donald object. If we would have different object with different name it would be attributes of those objects.

Now we can use that value in the methods. Like

```
print('quaaaaaaaack',self.v)
```

```
print('walk like a duck',self.v)
```

Now we create different object with different number in it.....`frank= duck(151)`.

All this happens because of self variables. When an object calls a method that self variables gets passed and that is reference to the object and all the things attached to objects.

Function first argument is always self and that argument doesn't get passed **explicitly**, its passed **implicitly** through the dot operator.

```
class Duck:
    def __init__(self,value):
        self.v=value

    def quack(self):
        print("quaaaaaaaack",self.v)

    def walk(self):
        print("walk like a duck",self.v)
```

```
Donald= Duck(58)
frank= Duck(51)
Donald.quack()
Donald.walk()
frank.quack()
frank.walk()
```

Another Example-

```
class BankAccount:
    def __init__(self):
```

```
        self.balance=0

    def deposit(self,amount):
        self.balance=self.balance+amount
        return self.balance

    def withdraw(self,amount):
        if amount>self.balance:
            print "Insufficient Funds"
        else:
            self.balance=self.balance-amount
            return self.balance

AC1213=BankAccount()
AC1214=BankAccount()

AC1213.deposit(3000)
AC1214.deposit(5000)

print AC1213.balance
print AC1214.balance

AC1213.withdraw(400000)
print AC1214.withdraw(3000)
```

## Python Inheritance

---

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more.

---

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new **class** with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

## Python Inheritance Syntax

```
class BaseClass:
```

```
    Body of base class
```

```
classDerivedClass(BaseClass):
```

Body of derived class

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

## Example- 1

```
#!/usr/bin/python

class Date(object):      # inherits from the 'object' class (will discuss shortly)
    def get_date(self):
        return '2014-10-13'

class Time(Date):         # inherits from the 'Date' class
    def get_time(self):
        return '08:13:07'

dt = Date()
print(dt.get_date())

tm = Time()
print(tm.get_time())
print(tm.get_date())    # found this method in the 'Date' class
```

As u can see we have 2 classes, date and time. Look closely at the class we see object in the parenthesis and date in the argument of time. What this means is that time inherits from date and date inherits from class called object.

Object is the built in class provided by python. Now take a look below, we created new date object(dt) and then we create a new time object(tm) .

```
61
62
63 # 1
64 class Date(object):
65     def get_date(self):
66         return '2015-9-11'
67 class Time(Date):
68     def get_time(self):
69         return '12:14:07'
70 dt=Date()
71 print dt.get_date()
72 tm=Time()
73 print tm.
74     get_date
75     get_time
76
```

Now look we are also able to call `get_date` on the time object, it is specifically because we place the date class name in the argument list of the time class definition. So they were able to call a date method on a time object.

## Another Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called `Polygon` defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side", i+1, "is", self.sides[i])
```

This class has data attributes to store the number of sides, `n` and magnitude of each side as a list, `sides`.

Method `inputSides()` takes in magnitude of each side and similarly, `dispSides()` will display these properly.

A triangle is a polygon with 3 sides. So, we can create a class called `Triangle` which inherits from `Polygon`. This makes all the attributes available in class `Polygon` readily available in `Triangle`. We don't need to define them again (code re-usability). `Triangle` is defined as follows.



```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self, 3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s * (s - a) * (s - b) * (s - c)) ** 0.5
        print('The area of the triangle is %0.2f' % area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1:3
Enter side 2:5
Enter side 3:4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle`, we were able to use them.

If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

### Another Example of Inheritance in Python

```
#!/usr/bin/python

class Animal(object):
    def __init__(self, name):
        self.name = name
    def eat(self, food):
        print '%s is eating %s.' % (self.name, food)

class Dog(Animal):
    def fetch(self, thing):
        print '%s goes after the %s!' % (self.name, thing)

class Cat(Animal):
    def swatstring(self):
        print '%s shreds the string!' % (self.name)

r = Dog('Rover')
f = Cat('Fluffy')

r.fetch('paper') # Rover goes after the paper!
f.swatstring() # Fluffy shreds the string!
r.eat('dog food') # Rover is eating dog food.
f.eat('cat food') # Fluffy is eating cat food.
r.swatstring() # AttributeError: 'Dog' object has no attribute 'swatstring'
```

Here we 've a class animals that inherits from object. Object is built in superclass from which all class inherits. Now see definition for dog and cat. As u can see they both inherits from animals, So what are animals attributes. There are 2 variables define in Animals. There are 2 methods that we find here. The fact that they are function definition doesn't make them any different, from any other variables.

Now look dog we 've one attribute and one method define fetch and in cat swatstring. Now go little beneath , we created a new dog object and created a new cat object. Since these are both pets we 've decided to give each animals a name. What happens when we construct the dog object, does \_\_init\_\_ gets called when we created a dog object, \_\_init\_\_ is not really any different from any other method. When we construct a dog object, the first that python does is look for \_\_init\_\_ in the dog class 'cause it is isn't there, python then checks to see if dog inherits from any other classes. When it see that dog inherits from animals, it looks for init attribute there. Sue there is init in animals, So python calls init there. We are able to avoid duplicating the code if we can arrange our classes in inheritance hierarchic manner.

## INHERITING THE CONSTRUCTOR

- `__init__` is like any other method; it can be inherited
- If a class does not have an `__init__` constructor, Python will check its parent class to see if it can find one
- As soon as it finds one, Python calls it and stops looking
- We can use the **`super()`** function to call methods in the parent class
- We may want to initialize in the parent as well as our own class

```
# /usr/bin/python
import random

class Animal(object):
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name):
        super(Dog, self).__init__(name)
        self.breed = random.choice(['Shih Tzu', 'Beagle', 'Mutt'])

    def fetch(self, thing):
        print '%s goes after the %s!' % (self.name, thing)

d = Dog('dogname')

print d.name
print d.breed
```

All animals a name and all dogs a particular breed. We called parent class constructor with `super` as shown in pic. So dog has its own `__init__` but the first thing that happen is we call `super`. **Super is built in function and it is designed to relate a class to its super class or its parent class.**

In this case we saying that get the super class of dog and pass the dog instance to whatever method we say here the constructor `__init__`.... So in another words we are calling `Animal.__init__` with the dog object. You may ask why we won't just say `Animal.__init__` with the dog instance, we could do this but if the name of animal class were to change, sometime in the future. What if we wanna rearrange the class hierarchy, so the dog inherited from another class. Using `super` in this case allows us to keep things modular and easy to change and maintain.

So in this example what we are trying to do is to use general in it functionality. In this case we wanna want dog to 've name just like all animals to 've a name but we also want to set its own breed which we are doing in this specialize `__init__` . So we are able to combine general `__init__` functionality with more specific functionality. This gives us opportunity to seprate common functionality from the specific functionality which can eliminate code duplication and relate class to one another in a way that reflects the system overall design.

## AnotherExample of Inheriting the constructor in Python

```
class Person(object):

    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):

    def __init__(self, first1, last1, staffnum):
        super(Employee,self).__init__(first1, last1)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + "," + self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x.Name())
print(y.GetEmployee())
```

## POLYMORPHISM :

It means many forms or shapes. It refers to the use of the same operation in objects of different classes.

It refers to 2 classes that 've methods of same name, we call it a common interface 'cause we can call the same method on either type of object.

POLYMORPHISM is the practice of using one object of one particular class as if it were another object of another class.

Now let's take a look how it is done in python. This is something at which python is really good at...

EXPLANATION- NOW we see duck quacks and dog bark....

These are two separate and distinct objects and they have two separate and distinct interfaces. If we want to use them polymorphically we need to make sure that they 've a common interface. And so the duck class, we can give it a bark and fur.....

```
class duck:
    def quack(self):
        print"quaaaaaaaaak"
    def walk(self):
        print"walks like a duck"
    def bark(self):
        print"the duck cannot bark"
    def fur(self):
        print"the duck has feathers"

class dog:
    def bark(self):
        print"woooooof"
    def fur(self):
        print"the dog has white and brown fur"
    def walk(self):
        print"walks like a dog"
    def quack(self):
        print"dog cannot quack"

donald=duck()
fido=dog()
for o in (donald,fido):
    o.quack()
    o.walk()
    o.bark()
    o.fur()
```

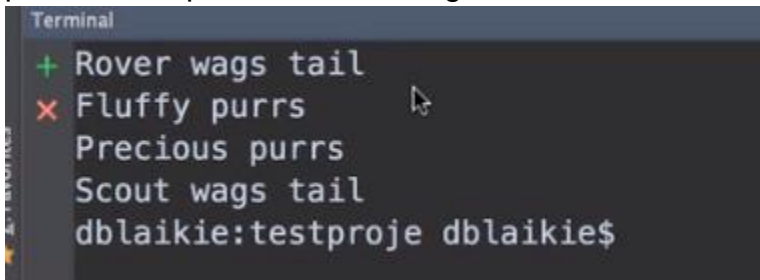
## Another Example of Polymorphism in Python

```
#!/usr/bin/python
class Animal(object):
    def __init__(self, name):
        self.name = name
    def eat(self, food):
        print '{0} eats {1}'.format(self.name, food)
class Dog(Animal):
    def fetch(self, thing):
        print '{0} goes after the {1}'.format(self.name, thing)
    def show_affection(self):
        print '{0} wags tail'.format(self.name)
class Cat(Animal):
    def swatstring(self):
        print '{0} shreds the string!'.format(self.name)
    def show_affection(self):
        print '{0} purrs'.format(self.name)
for a in (Dog('Rover'), Cat('Fluffy'), Cat('Precious'), Dog('Scout')):
    a.show_affection()
```

In this we 've added a show\_affection method in both the inherited class. Note that they aren't inherited methods but two entirely different methods with the same name. We doesn't see show\_affection in Animal class, and yet they appear in both dog and cat class that means they are independent methods.

```
for a in (Dog('Rover'), Cat('Fluffy'), Cat('Precious'), Dog('Scout')):
    a.show_affection()
```

NOW if we call show\_affection on a dog object then on cat object we see and we put new samples in Cat and dog



```
Terminal
+ Rover wags tail
x Fluffy purrs
Precious purrs
Scout wags tail
dblaikie:testproje dbaikie$
```

Even Though we call the same methods on each of the object, the object do entirely different things.

The convenient thing about this is as long as we know that we 've a animal object and as long as we understand that the contract is that the animal 'll 've show\_affection method, we don't need to care what type we dealing with, we can just call show affection and it 'll do the right thing. This feature is more of the technique than anything else note that it doesn't depend on magic functionality.

## Class variable:

A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

```
classEmployee:
    'Common base class for all employees'
    empCount=0

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

```
Employee.empCount+=1

def displayCount(self):
    print "Total Employee %d"%Employee.empCount

def displayEmployee(self):
    print "Name : ",self.name," , Salary: ",self.salary

"This would create first object of Employee class"
emp1 =Employee("Zara",2000)
"This would create second object of Employee class"
emp2 =Employee("Manni",5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d"%Employee.empCount
```

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

```
Name:Zara,Salary:2000
Name:Manni,Salary:5000
TotalEmployee2
```

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

```
emp1.age =7# Add an 'age' attribute.
emp1.age =8# Modify 'age' attribute.
del emp1.age # Delete 'age' attribute.
```

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

- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.

- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.

```
hasattr(emp1,'age')# Returns true if 'age' attribute exists  
getattr(emp1,'age')# Returns value of 'age' attribute  
setattr(emp1,'age',8)# Set attribute 'age' at 8  
delattr(emp1,'age')# Delete attribute 'age'
```

## Built-In Class Attributes

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

- **\_\_dict\_\_**: Dictionary containing the class's namespace.
- **\_\_doc\_\_**: Class documentation string or none, if undefined.
- **\_\_name\_\_**: Class name.
- **\_\_module\_\_**: Module name in which the class is defined. This attribute is "\_\_main\_\_" in interactive mode.
- **\_\_bases\_\_**: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

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

```
#!/usr/bin/python  
  
classEmployee:  
    'Common base class for all employees'  
    empCount=0  
  
    def __init__(self, name, salary):
```



```
self.name = name
self.salary= salary
Employee.empCount+=1

defdisplayCount(self):
print"Total Employee %d"%Employee.empCount

defdisplayEmployee(self):
print"Name : ",self.name," Salary: ",self.salary

print"Employee.__doc__:",Employee.__doc__
print"Employee.__name__:",Employee.__name__
print"Employee.__module__:",Employee.__module__
print"Employee.__bases__:",Employee.__bases__
print"Employee.__dict__:",Employee.__dict__
```

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

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

## Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

## Example

```
#!/usr/bin/python

classParent:# define parent class
defmyMethod(self):
    print'Calling parent method'

classChild(Parent):# define child class
defmyMethod(self):
    print'Calling child method'

c =Child()# instance of child
c.myMethod()# child calls overridden method
```

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

```
Calling child method
```

## Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders.

## Example

```
#!/usr/bin/python

classJustCounter:
    __secretCount=0

    def count(self):
        self.__secretCount+=1
        printself.__secretCount
```

```
counter=JustCounter()  
counter.count()  
counter.count()  
print counter.__secretCount
```

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

```
1  
2  
Traceback(most recent call last):  
File "test.py", line 12, in <module>  
print counter.__secretCount  
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object.\_\_className\_\_attrName*. If you would replace your last line as following, then it works for you –

```
.....  
print counter._JustCounter__secretCount
```

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

```
1  
2  
2
```

# Chapter-14

## REGULAR EXPRESSION

### REGULAR EXPRESSION

- A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- Regular expression is a module provided with python that allows you to do string searching and manipulation.
- It is used to do scrapping which is going to a webpage and searching through the HTML for very specific things.
- Used for searches, Find, Replace, validation....
- The basic idea with regexp is that they are way of searching for a pattern INSIDE a larger text. So very much like the search in MS word. You 've a little pattern and it 'll going to look into a huge text and find the first instance of that one.

### Why use regex :

- We can easily pull out what we want from huge volume of text.
- We can scrap webpages, email, address, phone no. , name or even pictures of jessicabeil or any other images you are interested in...

### Regular Expression Patterns

Except for control characters, (+ ? . \* ^ \$ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Pattern	Description
^	Matches beginning of line.
\$	Matches end of line.

.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a  b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.

(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Doesn't have a range.
(?! re)	Specifies position using pattern negation. Doesn't have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.

<code>\Z</code>	Matches end of string. If a newline exists, it matches just before newline.
<code>\z</code>	Matches end of string.
<code>\G</code>	Matches point where last match finished.
<code>\b</code>	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
<code>\B</code>	Matches nonword boundaries.
<code>\n, \t, etc.</code>	Matches newlines, carriage returns, tabs, etc.
<code>\1...\9</code>	Matches nth grouped subexpression.
<code>\10</code>	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

## The *match* Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function –

```
re.match(pattern,string, flags=0)
```

Here is the description of the parameters:

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below.

The `re.match` function returns a **match** object on success, **None** on failure. We use `group(num)` or `groups()` function of **match** object to get matched expression.

Match Object Methods	Description
<code>group(num=0)</code>	This method returns entire match (or specific subgroup num)
<code>groups()</code>	This method returns all matching subgroups in a tuple (empty if there weren't any)

## The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function:

```
re.search(pattern,string, flags=0)
```

Here is the description of the parameters:



Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern anywhere in the string.
flags	You can specify different flags using bitwise OR ( ). These are modifiers, which are listed in the table below.

The `re.search` function returns a **match** object on success, **none** on failure. We use `group(num)` or `groups()` function of **match** object to get matched expression.

Match Object Methods	Description
<code>group(num=0)</code>	This method returns entire match (or specific subgroup num)
<code>groups()</code>	This method returns all matching subgroups in a tuple (empty if there weren't any)

## Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default).

## Example

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

line="Cats are smarter than dogs";
```

```
matchObj=re.match(r'dogs', line,re.M|re.I)
if matchObj:
    print"match -->matchObj.group() : ",matchObj.group()
else:
    print"No match!!"

searchObj=re.search(r'dogs', line,re.M|re.I)
if searchObj:
    print"search -->searchObj.group() : ",searchObj.group()
else:
    print"Nothing found!!"
```

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

```
No match!!
search -->matchObj.group() : dogs
```

## Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

## Syntax

```
re.sub(pattern,repl,string)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

## Example

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

phone="2004-959-559 # This is Phone Number"

# Delete Python-style comments
num=re.sub(r'#..*$',"", phone)
```

```
print"Phone Num : ",num

# Remove anything other than digits
num=re.sub(r'\D','', phone)
print"Phone Num : ",num
```

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

```
PhoneNum:2004-959-559
PhoneNum:2004959559
```

## Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these –

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).

## Character classes

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

## Special Character Classes

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]

<code>\D</code>	Match a nondigit: <code>[^0-9]</code>
<code>\s</code>	Match a whitespace character: <code>[\t\r\n\f]</code>
<code>\S</code>	Match nonwhitespace: <code>[^\t\r\n\f]</code>
<code>\w</code>	Match a single word character: <code>[A-Za-z0-9_]</code>
<code>\W</code>	Match a nonword character: <code>[^A-Za-z0-9_]</code>

## Repetition Cases

Example	Description
<code>ruby?</code>	Match "rub" or "ruby": the y is optional
<code>ruby*</code>	Match "rub" plus 0 or more ys
<code>ruby+</code>	Match "rub" plus 1 or more ys
<code>\d{3}</code>	Match exactly 3 digits
<code>\d{3,}</code>	Match 3 or more digits
<code>\d{3,5}</code>	Match 3, 4, or 5 digits

Try out –

Now here we have made our own function to search for pattern

```
def find(pat,text):  
    match=re.search(pat,text)  
    if match: print match.group()  
    else: print "not found"
```

# every time i'm calling my function that takes 2 arguments. First is the pattern and second is the text.

```
find('iig','called piiig')
```

```
find('p.z','called piziig') # any three characters and then a g, and it'll search
```

```
find('p..g','called piiig') # It will print not found
```

# IN ORDER 2 SUCCEEDED ALL OF THE PATTERNS MUST MATCH...

# IMP.. SEARCH 'LL GO THROUGH LEFT TO RIGHT AND IT SATISFIES AS SOON IT #FINDS THE SOLUTION...

```
find('..g','called piiig much better:xyzg')
```

```
find('x..g','called piiig much better:xyzg')
```

```
find('..ii','calledpiiig much better:xyzgs')
```

```
#find(r'c\..l','c.lledpiiig much better:xyzgs')
```

#r= it means u do not do any special processing with back slashes, what ever i typed  
#just send it through absolute raw and uninterpreted.It is very useful for writing  
#regex, it frees us from worrying about the back slash processing...

```
find(r':\w\w\w','blah :123 :cat blah blah')
```

```
find(r':\d\d\d','blah :cat :123 blah blah')
```

```
find(r'\d\s\d\s\d','1 2 3')
```

# NOW IF WE 'VE SPACES varying B/W DIGITS

```
find(r'\d\s+\d\s+\d','1 2 3')
```

```
find(r':\w','blah blah :kitten blahblaha ')
```

```
find(r'.+ \@','blah blah :kitten@ 123 blahblaha ')
```

```
find(r':\w+', 'blah blah :kitten123&%$#*_ . blahblaha ')
```

# + is greedy it goes as far as it can and then it stops

```
find(r':.+', 'blah blah :kitten123&blahablaha ')
```

```
find(r'\S+', 'blahsdffghjkl;ghjklsd*; blah :kitten1233464756&=yattablahablaha ')
```

# EXAMPLE WITH E-MAIL

```
#find(r'\w+@\w+', 'blah hunk.a@gmail.com yatta @')
```

# it find the @ sign then a and it cannot go left in that 'cause the .(dot) doesn't

#count as back slash w character and like wise it gets the gmail but gets confines by .(dot)....

#[]= for showing set of character, here we can allow set of character in these bracket..

# in square bracket you don't need to back slash the .(dot). It understands that the dot

#inside the square bracket is just a dot...

#[] they are most convenient way, if there is some set of character you are lookig for....

```
find(r'[\w.]+@\w+', 'blah yoo.hunk.a@gmail.com yatta @')
```

```
find(r'[\w.]+@[ \w.]', 'blah yoo.hunk.a@gmail.com yatta @')
```

#If we apply .before hunk it would pick it up, 'cause we 've said the left of the

#@ sign.....

```
find(r'[\w.]+@[ \w.]', 'blah .hunk.a@gmail.com yatta @')
```

# 19, NOW IF WE SAY THE 1ST CHARACTER CAN'T BE A DOT IT MUST BE A WORD CHARACTER...

```
find(r'\w[\w.]+@[ \w.]', 'blah dsds.hunk.a@gmail.com yatta @')
```

# WE WANT TO CHOOSE THE USER NAME AND HOST NAME

```
m=re.search(r'([\w.]+)@([\w.]+)', 'blah hunk.a@gmail.com yatta @')
```

# by putting parenthesis in the regexp around the parts u cared about..Remember that

#parenthesis r not changing what its gonna match, it says that these r the 2 parts i

#cared about....

# now IF WE TYPE

```
#print m.group()
```

```
#print m.group(1)
```

```
#print m.group(2)
```

```
printre.findall(r'[\w.]+@[ \w.]', 'blah hunk.a@gmail.com yattafood@bar')
```

# findall= it just takes the pattern and rather than stoping at 1st match it just,

#continues and find all of the matches and reurns them to u the .(dot) group the whole

#text.....

# Ex- like f.read() to get that entire text in one string....so the pattern i was

#enjoying it because it saves me so much work..as like that we use (f.read) we can  
#feed the entire file into re.findall . I let it rip through the entire text and it pulls out  
#the thing that I want and just returns them to me as python list...

```
printre.findall(r'([\w.]+)@([\w.]+)', 'blah hunk.a@gmail.com yattafood@bar')
```

# it says if there are parenthesis in there instead of just returning the whole match  
#it says ohhh, there are two paren and i'll return tuples like two... Each tuples represent the  
#single match and tuples 'vegp on there.....