Introduction

It is a programming language with objects, modules, threads, exceptions and automatic memory management. The benefits of pythons are that it is simple and easy, portable, extensible, build-in data structure and it is an open source.

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

History of Python

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

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

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

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

Features of Python

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

- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.
- **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.
- Supports automatic **GarbageCollector**

<u>How Python is interpreted?</u>

Python language is an interpreted language. Python program runs directly from the source code. It converts the source code that is written by the programmer into an intermediate language, which is again translated into machine language that has to be executed. It never needs a seperate step of compilation. It generates its error during its runtime.

<u>Python Environment Setup</u>

```
python example.py

python –version  / python3 --version

python -V
```

<u>Interactive Mode</u>

When commands are read from a tty, the interpreter is said to be in ***interactive mode*.** In this mode it prompts for the next command with the ***primary prompt***, usually three greater-than signs

(>>>); for continuation lines it prompts with the ***secondary prompt***, by default three dots (...). Continuation lines are needed when entering a multi-line construct.

Memory Management

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if she regularly manipulates object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

Garbage Collection:

Python have an in-built garbage collector, which recycle all the unused memory and frees the memory and makes it available to the heap space. ie., Python deletes un-needed objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

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

Namespace:

In Python, every name introduced has a place where it lives and can be hooked for. This is known as namespace. It is like a box where a variable name is mapped to the object placed. Whenever the variable is searched out, this box will be searched, to get corresponding object.

Identifier

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers

- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
- Starting an identifier with a single leading underscore indicates that the identifier is private.

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
- Keywords cannot be used as identifiers.
- We cannot use special symbols like !, @, #, $, % etc. in our identifier.
- Identifier can be of any length.
- Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense.

Variables

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

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.

Multiple assignment of variables a,b = 12,78

Casting is when you convert a variable value from one type to another.

Accepting i/p from user : input() and raw_input()

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

del var

Keywords : It is a reserved word that cannot be used as an identifiers. All the Python keywords contain lowercase letters only.

Eg: if, else, elif, in, is, for,while, and, break, continue, pass, def, class etc...,

Datatypes in python

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

- ✓ Number
- ✓ String
- ✓ List
- ✓ Dictionary
- ✓ Tuple

Number: (int,long,float,complex)

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object. Number objects are created when you assign a value to them.

Eg : var1 = 10

Python supports four different **numerical types** −

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

Type conversion:

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.

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

Mathematical Functions

Python includes following functions that perform mathematical calculations.

Function	**Returns (description)**
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0 .
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	**x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
sqrt(x)	he square root of x for x > 0

<u>String</u>:

A string is a sequence of characters. Strings can be created by enclosing characters inside a single quote or double quotes.

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

Ex : str = ”abcd”.

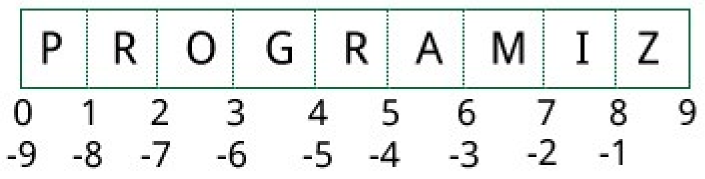
Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings. A string literal can span multiple lines, but there must be a backslash \ at the end of each line to escape the newline.

Python strings are "immutable" which means they cannot be changed after they are created

- ✓ string concatenation operator - (+)
- ✓ the repetition operator - (*)

<u>**Indexing and slicing (immutable object)**</u>

str[start:end] is the elements beginning at start and extending up to but not including end. Python strings are "immutable" which means they cannot be changed after they are created .Since strings can't be changed, we construct *new* strings as we go to represent computed values.



We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError.

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

String Functions

Method	**Functionality**	**Syntax**
title()	Capitalizes first letter of each word in string	str.title()
captalize()	Capitalizes first letter of string	str.capitalize()
lower()	Convert to lowercase	str.lower()
upper()	Convert to uppercase	str.upper()
startswith()	Returns True if string startswith given substring	str.startswith(‘a’)
endswith()	Returns False if string ends with given substring	str.endswith(‘a’)
count()	Count the number of chars	str.count(‘a’)
strip()	removing any character from both ends of a string.	str.strip()
lstrip()	removing any character from left ends of a string.	str.lstrip()
rstrip()	removing any character from right end of a string.	str.rstrip()
find()	find the sub-string in the string	str.find(‘x’)
len()	Returns the length of the given string	len(str)
swapcase()	Return a copy of str with opposite case	str.swapcase()
isdigit()	Returns True if all char are digit	str.isdigit()

isalpha()	Returns True if all char are alphabets	Str.isalpha()
isalnum()	Returns True if all char are alphanumeric	str.isalnum()
isspace()	Returns True if string contains space	str.space()
replace()	Replace a string with given string.	word.replace("Hello", "Goodbye")

Comment

A **comment** is a piece of code that is not run. In python, you make something a comment by putting a hash in front of it. A hash comments everything after it in the line, and nothing before it.

String Formatting Operator %

The % operator takes a printf-type format string on the left (%d int, %s string, %f/%g floating point), and the matching values in a tuple on the right (a tuple is made of values separated by commas, typically grouped inside parentheses).

Operators in Python

- ✓ Arithmetic Operators (+, -, *, /, %, //,**)
- ✓ Comparison Operators (==, <=, >=, <, >, !=)
- ✓ Assignment Operators (=, +=, -=, *=, /=, **=)
- ✓ Bitwiese Operators
- ✓ Logical Operators
- ✓ Membership Operators
    - ◆ Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples.
    - ◆ **in -** Evaluates to true if it finds a variable in the specified sequence and false otherwise.
    - ◆ **not in -** Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

- ✓ Identity Operators
    - ◆ Identity operators compare the memory locations of two objects.
    - ◆ **is** - Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
    - ◆ **is not** - Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

Difference between is and == in python

is will return **True** if two variables point to the same object, **==** if the objects referred to by the variables are equal.

is checks for *identity*. a is b is True iff a and b are the same object (they are both stored in the same memory address) And == checks for *equality*,

Control Statements

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

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

(i) If Statement :

An **if statement** consists of a boolean expression followed by one or more statements.

Syntax.

```
if <condition> :
        statements
```

(ii) If...Else Statement :

An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.

Syntax.

```
if <condition> :
        statements
else:
        statements
```

(iii) IF...elif....Else Statement :

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

Syntax.

```
if <condition1> :
        statements
elif condition2 :
        statements
elif condition3 :
        statements
```

(iv) Nested if:

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

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

Syntax.

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

Looping Statements

There may be a situation when you need to execute a block of code several number of times. A loop statement allows us to execute a statement or group of statements multiple times.

(i) for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

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

(ii) While loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

Syntax:

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

(iii) Nested loops

1. Nested while loop

Syntax:

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

2.Nested for loop

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.

Pass :

Pass means, no-operation Python statement, or in other words it is a place holder in compound statement, where there should be a blank left and nothing has to be written there.

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break :

Terminates the loop statement and transfers execution to the statement immediately following the loop.

ie., The break statement in Python terminates the current loop and resumes execution at the next statement.

Eg:

```
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!"
```

continue :

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. The **continue** statement in Python returns the control to the beginning of the 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. Loop does not terminate but continues on with the next iteration. The **continue** statement can be used in both *while* and *for* loops.

enumerate()

Enumerate is a built-in function of Python and adds a counter to an iterable. It allows us to loop over something and have an automatic counter. It returns an enumerate object. In our case that object is a list of tuples (immutable lists), each containing a pair of count/index and value.

Syntax:

enumerate(iterable object)

range():

It generates a list of numbers, which is generally used to iterate over with for loops. The range() function works a little bit differently between Python 2.x and 3.x under the hood, however the concept is the same. We'll get to that a bit later, however.

Python's range() Parameters

The range() function has two sets of parameters, as follows:

range(stop)

- stop: Number of integers (whole numbers) to generate, starting from zero. eg. range(3) == [0, 1, 2].

range([start], stop[, step])

- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence.

Note that:

- All parameters must be integers.
- All parameters can be positive or negative.
- range() (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. eg. The syntax to access the first element of a list is mylist[0]. Therefore the last integer generated by range() is up to, but not including, stop. For example range(0, 5) generates integers from 0 up to, but not including, 5.

xrange() :

- It generates a list of numbers, which is generally used to iterate over with for loops.
- xrange returns an xrange object ie., xrange doesn't actually generate a static list at run-time.
- It creates the values as you need them with a special technique called *yielding*. This technique is used with a type of object known as *generators*.
- It’s not in Python3.x
- And take parameters as same as of range().

Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type. Lists are collections of items where each item in the list has an assigned index value. A list is **mutable**, meaning you can change its contents.

Example : new_list = [1,2,3,4,5,6,7,8,9]

These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Accessing Values in Lists (Mutable Object)

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

Example 1 : new_list[0]

Example 2 : new_list[1:3]

Updating Lists

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

Example 1 : new_list[8] = 10

Example 2 : new_list.append(10)

Deletion

To delete entire list, use **del** command

Example 1 : **del** name_list

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting or the remove() method.

Example 1 : **del** new_list[1] //deletes 1st element

Example 2 : **del** new_list[6] //deletes 6th element

List operations

Operators and Methods	**Functionality**	**Syntax**
len()	Returns length of the list.	len(list)
+ operator	Concatenate two lists	list1 + list2
* operator	Repetition operator	list1 * list2

in operator	Membership operator . Returns True if item present in List.	3 in list2
append()	Appends object obj to list	list.append(obj)
count()	Returns count of how many times obj occurs in list	list.count(obj)
insert()	Inserts object obj into list at offset index	list.insert(index, obj)
remove()	Remove object from the list	list.remove(obj)
max()	Returns maximum element from the list	max(list)
min()	Returns minimum value from the list	min(list)
sort() or sorted()	sorts the list in place (do not return it)	list.sort() or sorted(list)
pop()	Returns the last element of the list	list.pop()
pop(index)	Returns the element of given index	list.pop(index)
clear()	method to empty a list	list.clear()
remove()	Remove item from the list	list.remove(item)
reverse()	Reverses objects of list in place	list.reverse()

List Comprehensions :

List comprehension is an elegant and concise way to create new list from an existing list in Python.

These lists have often the qualities of sets, but are not in all cases sets. The list comprehension always returns a result list.

A list comprehension consists of the following parts:

- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate

It can be used to construct lists in a very natural, easy way, like a mathematician is used to do.

In a general sense, a FOR loop works as:

```
for (set of values to iterate):
        if (conditional filtering):
                output_expression()
```

The same gets implemented in a simple LC construct in a single line as:

```
[ output_expression() for(set of values to iterate) if(conditional filtering) ]
```

LC will always return a result, whether you use the result or nor.

- ✓ The iteration and conditional expressions can be nested with multiple instances.
- ✓ Even the overall LC can be nested inside another LC.
- ✓ Multiple variables can be iterated and manipulated at same time.

Example 1: Generate a list from an existing list (resultant list must contain squares of odd numbers in the given list)

```
list1 = [1,2,3,4,5,6,7,8,9]
list2 = [x**2 for x in list1 if int(x)%2!=0]
```

Example 2 :{ x^2: x is a natural number less than 10 }

```
[x**2 for x in range(0,10)]
```

Example 3 : { x: x is a whole number less than 20, x is even }

```
[x for x in range(1,20) if x%2==0 ]
```

Example 4: { x: x is an alphabet in word ‘MATHEMATICS’, x is a vowel }

```
[x for x in 'MATHEMATICS' if x in ['A','E','I','O','U']]
```

Example 5: list comprehension using 2 for loops (iterating over 2 lists)

```
[x+y for x in ['Python ','C '] for y in ['Language','Programming']]
```

Tuples

A tuple is a sequence of immutable Python objects. Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

```
empty tuple  = ()
singleton tuple = (30,)
```

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. Tuples are immutable which means you cannot update or change the values of tuple elements.

Deletion

Removing individual tuple elements is not possible. To explicitly remove an entire tuple, just use the **del** statement.

Syntax: del tup1

Tuples are called immutable lists : Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Advantages of Tuple over List

Since, tuples are quite similiar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set(). It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the **set()** function without any argument. Sets are mutable. But since they are unordered, indexing have no meaning. We cannot access or change an element of set using indexing or slicing.

Update a set

```
setp.update([1,2,3,4,5,6])      // set updates with iterable object
setp.add(12)                    // set updates with single item
```

Set Operations

Consider A and B ae two sets then...

```
Union (AUB) ---> A | B
Intersection (An B) ---> A & B
Difference (A- B)  ----> A – B
Symmetric Diff  ------> A ^ B
```

You can't add a list to a set because lists are mutable, meaning that you can change the contents of the list after adding it to the set.

Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys. Frozensets can be created using the function frozenset().

Frozen set is just an immutable version of a Python set object. While, elements of a set can be modified at any time, elements of frozen set remains the same after creation.

Due to this, frozen sets can be used as key in Dictionary or as element of another set. But like sets, it is not ordered (the elements can be set at any index.

Dictionary

Python dictionary is an unordered collection of items and it contains **key : value** pair. 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.

- ✓ Empty Dictionary - {}
- ✓ Singleton Dictionary - {‘id’:56}

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

Dictionaries are unordered, so the order that the keys are added doesn't necessarily reflect what order they may be reported back.

A Python dictionary is a mapping of unique keys to values. Dictionaries are mutable, which means they can be changed. The values that the keys point to can be any Python value.

Accessing Value from Dict : To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example : dict1[‘id’]

Membership checking in dict: usin ‘in’ and ‘not in’

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.

```
dict1[‘id’] = 89        //updating
dict[‘name’] =’test’    //adding new entry
del dict[‘name’]        //deletion
```

Clearing all elements in dict : dict1.clear()

Deleting entire dictionary : del dict

1. More than one entry per key not allowed. Which means no duplicate key is allowed.
2. Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Built-in Dictionary Functions & Methods

Method	Purpose	Syntax
cmp()	Compare elements of both dict	cmp(dict1,dict2)
len()	Gives the total length of the dictionary	len(dict)
str()	String representation of dictionary	str(dict)
clear()	Removes all elements of dictionary	dict.clear()
fromkeys()	Create a new dictionary with keys from seq and values *set* to *value*	dict.fromkeys()
has_key()	Returns *true* if key in dictionary *dict*, *false* otherwise	dict.has_key(key)
items()	Returns a list of *dict*'s (key, value) tuple pairs	dict.items()
keys()	Returns a list of all the available keys in the dict.	dict.keys()
values()	Returns a list of all the available values in the dict.	dict.values()
update()	Adds dictionary *dict2*'s key-values pairs to *dict*	dict.update(dict2)
pop()	This method removes as item with the provided key and returns the value.	dict.pop(key)

Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python. It consists of an expression pair (key: value) followed by for statement inside curly braces {}. It may contain if statement for filter out items to form the new dictionary.

Dict comprehensions are just like list comprehensions, except that you group the expression using curly braces instead of square braces. Also, the left part before the for keyword expresses both a key and a value, separated by a colon.

A dictionary comprehension can optionally contain more for or if statements.

An optional if statement can filter out items to form the new dictionary.

Iterators

Generally, we use looping statements for iterating over sequences . So there are many types of objects which can be used with a for loop. These are called iterable objects. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

The Iteration Protocol :

The built-in function iter takes an iterable object and returns an iterator.

```
>>> x = iter([1, 2, 3])
>>> x
<list_iterator object at 0x7f28ad9797b8>
>>> x.__next__()
1
```

```
>>> x.__next__()
2
>>> x.__next__()
3
>>> x.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Each time we call the next method on the iterator gives us the next element. If there are no more elements, it raises a *StopIteration*.

In __next__() works ony in Python3.x. In python2 it can be implemented by using next(). Also Iterators are implemented as classes. Python iterator objects are required to support two methods while following the iterator protocol. __*iter*__ returns the iterator object itself. This is used in *for* and *in* statements. __*next*__ method returns the next value from the iterator. If there is no more items to return then it should raise *StopIteration* exception.

Generators:

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value. Each time the **yield** statement is executed the function generates a new value.

So a generator is also an iterator. When a generator function is called, it returns a generator object without even beginning execution of the function. When next method is called for the first time, the function starts executing until it reaches yield statement. The yielded value is returned by the next call.

Generator Expressions:

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

LAMBDA

The **lambda operator** or **lambda function** is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers. The general syntax of a lambda function is quite simple:

lambda argument_list: expression

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.

The following example of a lambda function returns the sum of its two arguments:

```
>>> f = lambda x, y : x + y
>>> f(1,1)
        2
```

map()

The "**map**" function transforms a given list into a new list by transforming each element using a rule.

It is used to apply a function to each element of a list or any other iterable.
Syntax: map(function, Python iterable)

%timeit, an in-built magic function of iPython notebook environment.

map() can take multiple lists as input arguments. This is very useful when we do vector mathematics. The map() function will take multiple lists, operate on them, and then return an output list. One thing to note is that all the lists should be of the same size because map() applies the function to corresponding elements in those lists.

```
my_list = [1, 5, 7, 8, 11]
new_list = [-4, 3, 2, -6, 5]
map(lambda x, y: x + y, my_list, new_list)
```

Comparison for loop,map(),LC

1. LC is faster because they involve storing values in a list.
2. LC is **fast and elegant in cases where simple expressions are involved**. But if complex functions are required, map and LC would perform nearly the same.
3. FOR-loop is bulkier in general, but it is fastest if storing is not required. So should be **preferred in cases where we need to simply iterate and perform operations.**

filter()

The "filter" function operates on a list and returns a subset of that list after applying the filtering rule.

The function **filter(function, list)** offers an elegant way to filter out all the elements of a list. The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.

Example 1: Filter even numbers in a list by using filter function. The list is: [1,2,3,4,5,6,7,8,9,10].

```
li = [1,2,3,4,5,6,7,8,9,10]
evenNumbers = filter(lambda x: x%2==0, li)
```

Example **2: Filter negative numbers from a list (list with in -5 to 5) by using filter function.**

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
```

reduce()

Reduce is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you wanted to compute the sum of a list of integers.

```
reduce(lambda x,y: x+y, [47,11,42,13])
```

The "**reduce**" function will transform a given list into a single value by applying a given function continuously to all the elements. It basically keeps operating on pairs of elements until there are no more elements left.

Zip()

The zip() function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and returns an iterator of tuples.

zip() Parameters:

The zip() function takes:

iterables - can be built-in iterables (like: list, string, dict), or user-defined iterables (object that has __iter__ method).

Syntax:

zip(*iterables)

Return Value from zip()

The zip() function returns an iterator of tuples based on the *iterable* object.

- ✓ If no parameters are passed, zip() returns an empty iterator []
- ✓ If a single iterable is passed, zip() returns an iterator of 1-tuples. Meaning, the number of elements in each tuple is 1.
- ✓ If multiple iterables are passed, ith tuple contains ith Suppose, two iterables are passed; one iterable containing 3 and other containing 5 elements. Then, the returned iterator has 3 tuples. It's because iterator stops when shortest iterable is exhaused.

Functions

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code. Functions may return a value to the caller, using the keyword- 'return' .

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

Functions in python are defined using the block keyword "def", followed with the function's name as the block's name. Functions may also receive arguments (variables passed from the caller to the function).

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.

The return statement is used to exit a function and go back to the place from where it was called.

Calling a function

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

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Pass by reference:

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. Here we are maintaining reference of the passed object and appending values in the same object.

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.

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. Y

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.

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

LAMBDA Function

In **Python**, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in **Python** anonymous functions are defined using the **lambda** keyword. Hence, anonymous functions are also called **lambda** functions.

Python supports a style of programming called *functional programming* where you can pass functions to other functions to do stuff.

The **lambda operator** or **lambda function** is a way to create small anonymous functions, i.e. functions without a name. These functions are throw-away functions, i.e. they are just needed where they have been created. Lambda functions are mainly used in combination with the functions filter(), map() and reduce(). The lambda feature was added to Python due to the demand from Lisp programmers.

The general syntax of a lambda function is quite simple:
lambda argument_list: expression
The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments. You can assign the function to a variable to give it a name.

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.
The following example of a lambda function returns the sum of its two arguments:

```
>>> f = lambda x, y : x + y
>>> f(1,1)
```

Decorators :

Decorators allow you to make simple modifications to callable objects like functions, methods, or classes. a decorator takes in a function, adds some functionality and returns it.

A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. Doecorator acts as a wrapper for the original function.

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

Use of Decorators

1. Avoid code duplication
2. Cluttering main logic of function with additional functionality.

Common usecase of decorator include timing ang loging purpose.

Object Oriented Programming Concepts : -

Class

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

```
class MyClass:
    variable = "blah"

    def function(self):
        print("This is a message inside the class.")

myobjectx = MyClass()
myobjectx.variable
myobjectx.function()
```

Constructors **:**

Class functions that begins with double underscore (__) are called special functions as they have special meaning like __init__() function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP).

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance.

The automatic destruction of unreferenced objects in Python is also called garbage collection.

self keyword:

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name self.

In the __init__ **method**, **self** refers to the newly created object; in other class **methods**, it refers to the instance whose **method** was called.
Example:

```
class Employee:
        'Common base class for all employees'
        empCount = 0

        def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

        def displayCount(self):
                print "Total Employee %d" % Employee.empCount

        def displayEmployee(self):
        print "Name : ", self.name,  ", Salary: ", self.salary

emp1 = Employee("Zara", 2000)            //object creation
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Class & instance variable

instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

  kind = 'canine'        # class variable shared by all instances
  def __init__(self, name):
     self.name = name    # instance variable unique to each instance
```

```
d = Dog('Fido')
e = Dog('Buddy')
d.kind                  # shared by all dogs
            'canine'
e.kind                  # shared by all dogs
            'canine'
d.name                  # unique to d
            'Fido'
e.name                  # unique to e
            'Buddy'
```

Mutable objects as instance variable :

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.tricks
['roll over']
e.tricks
['play dead']
```

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.

```
class JustCounter:
  __secretCount = 0

  def count(self):
    self.__secretCount += 1
    print self.__secretCount

counter = JustCounter()          //object creation
counter.count()                  //var visible to class
counter.count()                  //var visible to class
print counter.__secretCount      //hidden variable
```

You can access such attributes as *object._className__attrName*.

```
print counter._JustCounter__ secretCount
```

Inheritance

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

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**. The name BaseClassName must be defined in a scope containing the derived class definition.

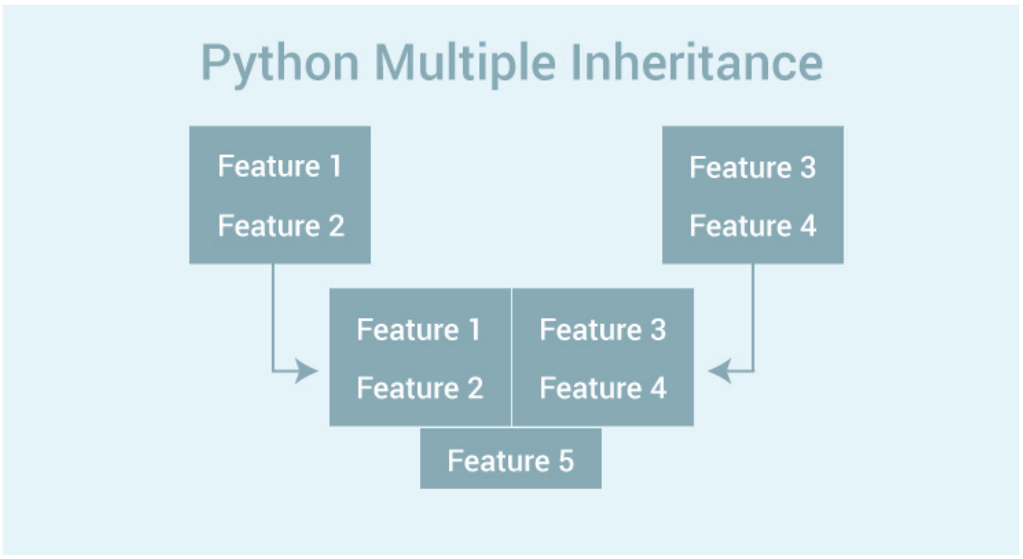
```
class BaseClass:
        Body of base class
class DerivedClass(BaseClass):
        Body of derived class
```

1. ***Single inheritance :***

Single inheritance enables a derived class to **inherit** properties and behavior from a **single** parent class. It allows a derived class to **inherit** the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code.

2. ***Multiple inheritance***

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can **inherit** characteristics and features from more than one parent object or parent class.



In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single inheritance.

Super keyword

- Working with Multiple Inheritance
- Allows us to avoid using base class explicitly

The super() builtin returns a proxy object, a substitute object that has ability to call method of the base class via delegation. Ability to reference base object with super() is called indirection.

Multilevel Inheritance in Python

You can inherit a derived class from another derived class. This is known as multilevel inheritance. In Python, multilevel inheritance can be done at any depth.

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

An example with corresponding visualization is given below.

```
class Base:
        pass

class Derived1(Base):
        pass

class Derived2(Derived1):
        pass
```

Overloading

Operator overloading: work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

Method overloading: In Python you can define a method in such a way that there are multiple ways to call it. Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as *method overloading.*

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()
```

Abstract Class

Abstract Base Classes (ABCs) ensure that derived classes implement particular methods from the base class.

abc module provides the infrastructure for defining abstract base classes .

abc module allows to enforce that a derived class implements a particular method using a special **@abstractmethod** decorator on that method.

In python 2.7

```
from abc import ABCMeta, abstractmethod
```

```
class Abstract:
    __metaclass__ = ABCMeta

    @abstractmethod
    def foo(self):
        pass
```

Example:

```
from abc import ABCMeta, abstractmethod

class Animal:
    __metaclass__ = ABCMeta

    @abstractmethod
    def say_something(self): pass

class Cat(Animal):
    def say_something(self):
        return "Miauuu!"

a = Animal()      //results error becoz can’t create an objct for abstract class
```

We can’t instantiate an abstract class. An abstract method can also have an implementation, but it can only be invoked with super from a derived class.

Why we need Abstract classes

Abstract base classes are a form of interface checking more strict than individual hasattr() checks for particular methods. By defining an abstract base class, you can define a common API for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, such as with plugins to an application, but can also aid you when working on a large team or with a large code-base where keeping all classes in your head at the same time is difficult or not possible.

- ✓ Abstract Base Classes (ABCs) ensure that derived classes implement particular methods from the base class at instantiation time.
- ✓ Using ABCs can help avoid bugs and make class hierarchies easier to maintain.

POLYMORPHISM

Polymorphism is based on the greek words Poly (many) and morphism (forms). We will create a structure that can take or use many forms of objects. Polymorphism can be referred as the process of taking many forms by a same object.

1. Method overloading
    In Python you can define a method in such a way that there are multiple waysto call it. Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters. This is known as *method overloading.*

```
class Human:
    def sayHello(self, name=None):

        if name is not None:
            print 'Hello ' + name
        else:
            print 'Hello '

# Create instance
obj = Human()

# Call the method
obj.sayHello()

# Call the method with a parameter
obj.sayHello('Guido')
```

2. Method overriding

Polymorphic behaviour allows you to specify common methods in an "abstract" level, and implement them in particular instances. **Overriding** is the ability of a class to change the implementation of a **method** provided by one of its ancestors. **Overriding** is a very important part of OOP since it is the feature that makes inheritance exploit its full power.

In Python method overriding occurs simply defining in the child class a method with the same name of a method in the parent class. When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come in play.

```
class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

```
Output :
   Calling child method
```

Class method:-

A class method is a method that is bound to a class rather than its object. It doesn't require creation of a class instance, much like staticmethod. The classmethod() method returns a class method for the given function.We can use the @classmethod decorator for classmethod definition. The class method is always attached to a class with first argument as the class itself *cls*. The class method can be called both by the class and its object.
@classmethod. This is called a decorator for converting method() to a class method as classmethod().

The difference between a static method and a class method is:

- ✓ Static method knows nothing about the class and just deals with the parameters
- ✓ Class method works with the class since its parameter is always the class itself.

Packages

We use a well-organized hierarchy of directories for easier access. Python has packages for directories and modules for files. This makes a project (program) easy to manage and conceptually clear. This makes a project (program) easy to manage and conceptually clear. We can import modules from packages using the dot (.) operator.

Files

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. Each line is terminated with a special character, called the EOL or **End of Line** character.

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. 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.

Open ( ) Function

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

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

The *close()* Method

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

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

Syntax : fileobject.close()

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.

Write()

The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text. The write() method does not add a newline character ('\n') to the end of the string. If the specified file not exists, then it will create a new one. Instead of using “w” we can use **w+** or **wb** or **wb+**

Example :

```
file_obj = open(“test.txt”, “w”)   //open file in write mode

file_obj .write( "Python is a great language.\nYeah its great!!\n");

file_obj .close()
```

Read data from a file()

To read from a file, first step is to open the file read (r) mode. Instead of using “r” mode, we can use r+ or rb or rb+.

file.read(n) - This method reads n number of characters from the file, or if n is blank it reads the entire file.

file.readline(n) - This method reads an entire line from the text file.

The read*()* method reads the contents of the file. You can assign the result to a variable. It is important to note that Python strings can have binary data and not just text. To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

We tell Python to read the entire file with read() because we did not provide any arguments.

Syntax : file_obj.read()

readline()

It will reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline. Each time you run the method, it will return a string of characters that contains a single line of information from the file.

Example 1 : print file.readline() //will return the first line of the file.

Example 2 : print file.readline(3) //will return the third line of the file.

readlines()

It will reads entire lines from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.

Looping over

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code.

`f.tell()` :- Returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.

`f.seek()` :- To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

Handling JSON data with files

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

`dump()` : - simply serializes the object to a file

`load()` : - simply deserializes the object

Append data to a file : - To append data to an existing file use the command open("Filename", "a").

Delete a file : - To delete a file we use remove() Syntax is : **os.remove(filename).**

Rename a file : - *rename()* method takes two arguments, the current filename and the new filename. Syntax is : **os.rename(old_filename, new_filename).**

"with" statement :-

It is possible to open a file using ***with*** statement. It will automatically close the file after the nested block of code. The advantage of using a `with` statement is that it is guaranteed to close the file no matter *how* the nested block exits. If an exception occurs before the end of the block, it will close the file before the exception is caught by an outer exception handler. If the nested block were to contain a `return` statement, or a `continue` or `break` statement, the `with` statement would automatically close the file in those cases, too.

```
with open("welcome.txt") as file:

    data = file.read()
```

Exception :

An exception is an error that happens during the execution of a program. That is, Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called ***exceptions*** and are not unconditionally fatal. Exceptions are known to non-programmers as instances that do not conform to a general rule. The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule".

Exception Handling in Python :

Exception handling is a construct in some programming languages to handle or deal with errors automatically. Many programming languages like C++, Objective-C, PHP, Java, Ruby, Python, and many others have built-in support for exception handling.

When exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash. In Python, exceptions can be handled using a try statement. A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause. It is up to us, what operations we perform once we have caught the exception.

Error Handling :

Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

Try ......except

A **`try`** statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same `try` statement. An except clause may name multiple exceptions as a parenthesized tuple. Exceptions are caught by **except** clause.

It's possible to "create custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is “compatible” with the exception. An object is compatible with an exception if it is the class or a base class of the exception object or a tuple containing an item compatible with the exception.

When a matching except clause is found, the exception is assigned to the target specified after the `as` keyword in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. When an exception has been assigned using `as target`, it is cleared at the end of the except clause.

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

The optional `else` clause is executed if and when control flows off the end of the `try` clause. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

If `finally` is present, it specifies a 'cleanup' handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `finally` clause executes a `return` or `break` statement, the saved exception is discarded. This clause is executed no matter what, and is generally used to release external resources.

```
except E as N:
    foo
```

```
try:
        code
except (RuntimeError, TypeError, NameError):
...     pass
```

<u>try....except..else</u>

The `try` … `except` statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

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**. The `raise` statement allows the programmer to force a specified exception to occur. We can also optionally pass in value to the exception to clarify why that exception was raised.

User defined Exceptions :

Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Threads

A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system. Threads are usually contained in processes. Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes. Every process has at least one thread, i.e. the process itself.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Advantages of Threading:

- Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be executed truly concurrent.
- A program can remain responsive to input. This is true both on single and on multiple CPU
- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

Threads in Python

There are two modules which support the usage of threads in Python:

1. thread
2. threading

The module "thread" treats a thread as a function, while the module "threading" is implemented in an object oriented way, i.e. every thread corresponds to an object.

thread Module

from thread import start_new_thread,

```
thread.start_new_thread ( function, args[, kwargs] )
```

The method call returns immediately and the child thread starts and calls function with the passed list of *args*. When function returns, the thread terminates. Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

Threading Module

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

- **threading.activeCount():** Returns the number of thread objects that are active.
- **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

- **run():** The run() method is the entry point for a thread.
- **start():** The start() method starts a thread by calling the run method.
- **join([time]):** The join() waits for threads to terminate.
- **isAlive():** The isAlive() method checks whether a thread is still executing.
- **getName():** The getName() method returns the name of a thread.
- **setName():** The setName() method sets the name of a thread.

To implement a new thread using the threading module, you have to do the following −

- Define a new subclass of the *Thread* class.
- Override the *__init__(self [,args])* method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()* method.

<u>Multithreading</u> :-

More than one thread can exist within the same process. These threads share the memory and the state of the process. In other words: They share the code or instructions and the values of its variables.

Running several threads is similar to running several different programs concurrently, but with the following benefits −

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
- Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.
- Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time. This statement holds true for both multiprocessor or single processor systems.
- All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

<u>Thread Synchronization :-</u>

The <*threading*> module has built in functionality to implement locking that allows you to synchronize threads. Locking is required to control access to shared resources to prevent corruption or missed data.

You can call *Lock()* method to apply locks, it returns the new lock object. Then, you can invoke the *acquire(blocking)* method of the lock object to enforce threads to run synchronously.

The optional *blocking* parameter specifies whether the thread waits to acquire the lock.

- In case, *blocking* is set to zero, the thread returns immediately with a zero value if the lock can’t be acquired and with a 1 if the lock was acquired.
- In case, *blocking* is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the lock object is used to release the lock when it is no longer required. Python’s built-in data structures such as lists, dictionaries are thread-safe as a side-effect of having atomic byte-codes for manipulating them. Other data structures implemented in Python or basic types like integers and floats, don’t have that protection. To guard against simultaneous access to an object, we use a *Lock* object.