



Let's Solve



Celebrating 20 Years

Python Programming



Larsen & Toubro
Group Company

What are we going to learn?

- Introduction To Python Fundamentals
- Data Types, Variables and Operators
- Conditional statements/Control Structures
- Basic Data Structures
- Functions
- Introduction to OOP
- Handling Exceptions
- File handling
- Database connectivity

Introduction

■ Brief History of Python :

- Invented by Guido Van Rossum (CWI,Amsterdam,1991)
- Named after “Monty Python’s Flying Circus “ 1969-1974
- Now owned by the Python Software Foundation (PSF)
- Derived from ABC, Modula-3, Lisp and “C” languages

Scripting language or Programming language

▪ Scripting Language

- "normal users" can write short programs in that language to automate certain tasks in a defined environment
- It is an interpreted language
- Eg : Javascript / perl

Python - A scripting language which takes on the advantages of programming too

▪ Programming Language

- used to write complex software that is not changed often
- It is a compiled and interpreted language
- Eg : Java

Python Features

- Scripting language
- Interpreted ,Interactive
- Elegant Syntax and dynamic typing
- Rapid Application Development
- Object Oriented
- Portable
- Powerful
- Cross platform
- Easy to Learn and Use

Python vs. other languages

JAVA



```
// Hello World in Java
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

C++



```
// Hello World in C++
#include <iostream.h>
Main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

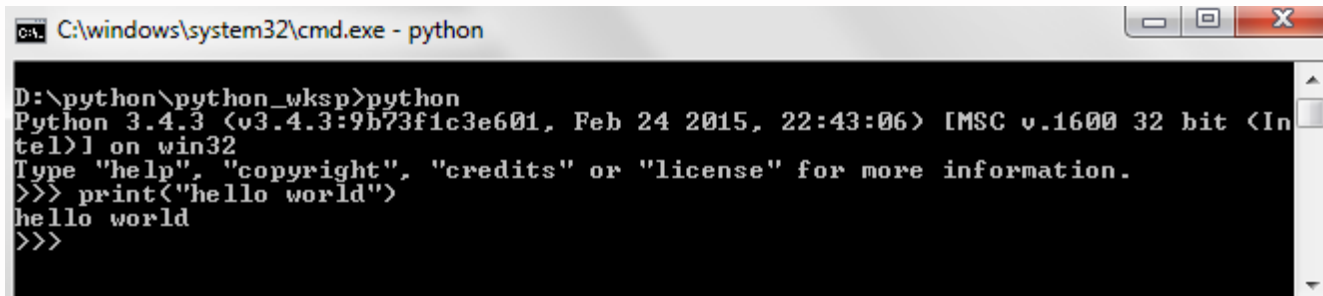
PYTHON



```
# Hello World in Python
print 'Hello World!'
```

Python environment

- Python programming can be done using a shell like Power Shell or command Prompt

A screenshot of a Windows Command Prompt window. The title bar shows the path 'C:\windows\system32\cmd.exe - python'. The command prompt shows the user running 'python' at the directory 'D:\python\python_wksp'. The output displays the Python 3.4.3 version information and the prompt 'tel>'. The user then enters 'print("hello world")' and the output 'hello world' is displayed.

```
C:\windows\system32\cmd.exe - python

D:\python\python_wksp>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>>
```

- The various IDE's are also available for Python Programming
 - Eclipse
 - PyCharm
 - Boa Constructot
 - PythonWin
 - IDLE
 - etc

Syntax

- Spaces/Indentation marks the block
 - Traditional 4 spaces
- No curly braces
- No ; as terminator
- Blocks are followed by :
- No Type declaration for data
- No new keyword
- .py file acts as a module

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain
 - Letters
 - Numbers
 - Underscores

Example :

- bob, Bob, _bob, _2_bob_, bob_2 , BoB
- Keywords / Reserved words cannot be used as names

Whitespace

- Whitespace is having significance in Python: especially indentation and placement of newlines
 - Use a newline to end a line of code
 - ✓ Not a semicolon like in C++ or Java
 - ✓ Use '\n' when must go to next line prematurely
 - No braces { } to mark blocks of code in Python
 - ✓ Use consistent indentation instead
 - ✓ The first line with a new indentation is considered outside of the block

Indenting Code

- Blocks of code are defined by indentation
 - ✓ Indenting starts a block and unindenting ends it
 - ✓ No need for explicit braces, brackets or keywords
 - ✓ Whitespace is significant
 - ✓ Code blocks are started by a ':'

```
def fib(n):  
    print 'n =', n  
    if n > 1:  
        return n * fib(n - 1)  
    else:  
        print 'end of the line'  
        return 1
```

Data Types

▪ Scalars:

- ✓ Integer -> 2323, 3234L
- ✓ Float -> 32.3, 3.1E2
- ✓ Boolean -> True, False

▪ Aggregate Types

- ✓ Complex -> $3 + 2j$, $1j$
- ✓ String -> "abc", 'abc'
- ✓ List -> l = [1,2,3]
- ✓ Dictionaries -> d = {'hello' : 'there', 2 : 15}
- ✓ Tuple -> t = (1,2,3)
- ✓ File
- ✓ Set

Variables

- No need to declare variables as specific type (ex: integer, float, string)
 - ✓ Example:
 - A, B = 3, 'cat'
- Need to assign /initialize variables
 - ✓ Use of uninitialized variable raises exception
- Variables are not strongly typed. A variable's type is dynamic, and will changed whenever it is reassigned
 - if friendly:
 - greeting = "hello world"
 - else:
 - greeting = 12**2
 - print greeting
- Variables are reference to Objects
- Everything in Python is a object
 - ✓ Functions, Classes, Modules

Operators

▪ Arithmetic/Logical :

- ✓ Assignment
- ✓ Comparison
- ✓ Logical
- ✓ Others

: =

: ==

: and, or, not (no symbols &&, || ,!)

: +, -, *, /, %, <, <=, >=, >, ==, !=

▪ Identity

: is, is not

▪ Membership

: in, not in

▪ Bitwise

: | ^ & ~

▪ Boolean

: true, false

- ✓ '0' and 'None' are false
- ✓ It is true other than '0' and 'None'
- ✓ True and False are aliases for 1 and 0 respectively



Object Oriented Python

OOPS

- Python is an object oriented programming language.
- Unlike procedure oriented programming, in which the main emphasis is on functions, object oriented programming stress on objects.
- Object is simply a collection of data (variables) and methods (functions) that act on those data.
- Four major principles of object-orientation
 - Encapsulation
 - Data Abstraction
 - Polymorphism
 - Inheritance

Class

- Class is a blueprint for the object.
- We can create many objects from a class.
- An object is also called an instance of a class and the process of creating this object is called instantiation.

Syntax

- Define a class using the keyword class.

```
class Duck:  
    pass
```

Creating object of a class Book

- `donald = Duck()`

Constructor in a class

- Class functions that begins with double underscore (__) are called special functions as they have special meaning.
- The __init__() function gets called whenever a new object of that class is instantiated.
- This type of function is also called constructors in Object Oriented Programming
- We normally use it to initialize all the variables.
- The self parameter is passed to the constructor by default.
- Constructor can consist of other parameters to initialize the instance variables of a class.

Instance Variables

- Instance variables are for data unique to each instance
- When you assign a value to a variable using self the instance variable get created

```
def __init__(self,name):  
    self.name = name
```

Private, public and protected members

Naming	Type	Meaning
name	Public	These attributes can be freely used inside or outside of a class definition.
_name	Protected	Protected attributes should not be used outside of the class definition, unless inside of a subclass definition.
__name	Private	This kind of attribute is inaccessible and invisible. It's neither possible to read nor write to those attributes, except inside of the class definition itself.

Getter and Setters

- Getters and setters are used in many object oriented programming languages to ensure the principle of data encapsulation.
- They are known as mutator methods as well.
- Getter for retrieving the data and the setter for changing the data.
- According to this principle, the attributes of a class are made private to hide and protect them from other code.
- For every private attribute of our class a getter and a setter can be defined

```
def setColor(self,color):
```

```
    self.__color=color
```

```
def getColor(self):
```

```
    return self.__color
```

Destructors

- The method `__del__` is used as a destructor.
- It is called when the instance is about to be destroyed and if there is no other reference to this instance.
- If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.
- Syntax to create a destructor is as follows:

```
def __del__(self):  
    print ("Object has been destroyed")
```

Inheritance

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



Basic Data Structures

Tuples

- A tuple is an immutable list
- A tuple is defined analogously to lists, except that the set of elements is enclosed in parentheses instead of square brackets.
- The rules for indices are the same as for lists. Once a tuple has been created, you can't add elements to a tuple or remove elements from a tuple.
- Where is the benefit of tuples?
 - Tuples are faster than lists.
 - If you know that some data doesn't have to be changed, you should use tuples instead of lists, because this protect your data against accidental changes to these data.
 - Tuples can be used as keys in dictionaries, while lists can't.

Create Tuples

- # empty tuple

```
my_tuple = ()
```

- # tuple having integers

```
my_tuple = (1, 2, 3)
```

- # tuple with mixed datatypes

```
my_tuple = (1, "Hello", 3.4)
```

- # nested tuple

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
```

- # tuple can be created without parentheses also called tuple packing

```
my_tuple = 3, 4.6, "dog"
```

- # tuple unpacking is also possible

```
a, b, c = my_tuple
```

Indexing Tuples

- `my_tuple = ['p','e','r','m','i','t']`
- `my_tuple[0]` 'p'
- `my_tuple[5]` 't'
- `my_tuple[6]` # index must be in range
- `my_tuple[2.0]` # index must be an integer
- `n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))`
- `n_tuple[0][3]` # nested index 's'
- `n_tuple[1][1]` # nested index 4
- `n_tuple[2][0]` # nested index 1

Slicing Tuples

- `my_tuple = ('p','r','o','g','r','a','m','i','z')`
- `my_tuple[1:4]` # elements 2nd to 4th ('r', 'o', 'g')
- `my_tuple[:7]` # elements beginning to 2nd ('p', 'r')
- `my_tuple[7:]` # elements 8th to end ('i', 'z')
- `my_tuple[:]` # elements beginning to end

Tuple Methods

Method	Description
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Return the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

List

- A list is created by placing all the items (elements) inside a square bracket [], separated by commas.
- It can have any number of items and they may be of different types (integer, float, string etc.).
- A list can even have another list as an item. These are called nested list.
- # empty list `my_list = []`
- # list of integers `my_list = [1, 2, 3]`
- # list with mixed datatypes `my_list = [1, "Hello", 3.4]`
- # nested list `my_list = ["mouse", [8, 4, 6]]`
- Slicing, Indexing, Changing and deleting are same as tuples.

List methods

These methods are accessed as `list.method()`.

Method	Description
<code>append(x)</code>	Add item x at the end of the list
<code>extend(L)</code>	Add all items in given list L to the end
<code>insert(i,x)</code>	Insert item x at position i
<code>remove(x)</code>	Remove first item that is equal to x , from the list
<code>pop([i])</code>	Remove and return item at position i (last item if i is not provided)
<code>clear()</code>	Remove all items and empty the list
<code>index(x)</code>	Return index of first item that is equal to x
<code>count(x)</code>	Return the number of items that is equal to x
<code>sort()</code>	Sort items in a list in ascending order
<code>reverse()</code>	Reverse the order of items in a list
<code>copy()</code>	Return a shallow copy of the list

Built in functions are same as tuples

Set

- Set is an unordered collection of items.
- Every element is unique (no duplicates) and must be immutable.
- However, the set itself is mutable (we can add or remove items).
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- Curly braces or the `set()` function can be used to create sets.
- Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary

Create Set

- # set of integers

```
my_set = {1, 2, 3}
```

- # set of mixed datatypes

```
my_set = {1.0, "Hello", (1, 2, 3)}
```

- # set do not have duplicates

```
{1,2,3,4,3,2} {1, 2, 3, 4}
```

- # set cannot have mutable items

```
my_set = {1, 2, [3, 4]}
```

- # but we can make set from a list

```
set([1,2,3,2]) {1, 2, 3}
```

Changing or deleting Set

- Set does not support indexing or slicing as it is unordered
- We can add single elements using the method `add()`.
- Multiple elements can be added using `update()` method.
- The `update()` method can take tuples, lists, strings or other sets as its argument.
- In all cases, duplicates are avoided.
- `my_set = {1,3}`
- `my_set[0]` #'set' object does not support indexing
- `my_set.add(2)`
- `my_set.update([4,5], {1,6,8})`
- Removing Elements from a Set
- `my_set.discard(4)`

Set Methods

Method	Description
<code>add()</code>	Add an element to a set
<code>clear()</code>	Remove all elements from a set
<code>copy()</code>	Return a shallow copy of a set
<code>difference()</code>	Return the difference of two or more sets as a new set
<code>difference_update()</code>	Remove all elements of another set from this set
<code>discard()</code>	Remove element from set if it is a member..Do nothing if element not in set
<code>intersection()</code>	Return the intersection of two sets as a new set
<code>intersection_update()</code>	Update the set with the intersection of itself and another
<code>isdisjoint()</code>	Return True if two sets have a null intersection
<code>issubset()</code>	Return True if another set contains this set
<code>issuperset()</code>	Return True if this set contains another set
<code>pop()</code>	Remove and return an arbitrary set element. Raise <code>KeyError</code> if the set empty
<code>remove()</code>	Remove an element from a set. If element is not a member, raise a <code>KeyError</code>
<code>symmetric_difference()</code>	Return the symmetric difference of two sets as a new set
<code>symmetric_difference_update()</code>	Update a set with the symmetric difference of itself and another
<code>union()</code>	Return the union of sets in a new set
<code>update()</code>	Update a set with the union of itself and others

Dictionary

- Unordered collection of items.
- Has a key: value pair.
- Indexed by *keys* - immutable type; strings and numbers.
- Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key.
- Lists can't be used as keys.
- Optimized to retrieve values when the key is known.
- Main operations are
 - Storing a value with some key and extracting the value given the key.
 - Delete a key:value pair with `del`.
 - Store using a key already in use, the old value associated with that key is forgotten.
- It is an error to extract a value using a non-existent key.
- `list(d.keys())` - returns a list of all the keys in arbitrary order (if you want it sorted, just use `sorted(d.keys())` instead).
- Check if a single key is in the dictionary, use the `in` keyword.

Dictionary Methods

Method	Description
<code>clear()</code>	Remove all items form the dictionary.
<code>copy()</code>	Return a shallow copy of the dictionary.
<code>fromkeys(seq[,v])</code>	Return a new dictionary with keys from <i>seq</i> and value equal to <i>v</i> (defaults to <i>None</i>).
<code>get(key[,d])</code>	Return the value of <i>key</i> . If <i>key</i> doesnot exit, return <i>d</i> (defaults to <i>None</i>).
<code>items()</code>	Return a new view of the dictionary's items (key, value).
<code>keys()</code>	Return a new view of the dictionary's keys.
<code>pop(key[,d])</code>	Remove the item with <i>key</i> and return its value or <i>d</i> if <i>key</i> is not found. If <i>d</i> is not provided and <i>key</i> is not found, raises <i>KeyError</i> .
<code>popitem()</code>	Remove and return an arbitrary item (key, value). Raises <i>KeyError</i> if the dictionary is empty.
<code>setdefault(key[,d])</code>	If <i>key</i> is in the dictionary, return its value. If not, insert <i>key</i> with a value of <i>d</i> and return <i>d</i> (defaults to <i>None</i>).
<code>update([other])</code>	Update the dictionary with the key/value pairs from <i>other</i> , overwriting existing keys.
<code>values()</code>	Return a new view of the dictionary's values



Exception Handling

Exception

- An error that happens during the execution of a program.
- 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.
- 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: you will soon learn how to handle them in Python programs.
- Most exceptions are not handled by programs, however, and result in error messages as shown here

10 * (1/0)

ZeroDivisionError: division by zero

4 + spam*3

NameError: name 'spam' is not defined

'2' + 2

TypeError: Can't convert 'int' object to str implicitly

Raising exception

- The raise statement allows the programmer to force a specified exception to occur.
- For example:

```
raise NameError('HiThere')
```

- The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).
- If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception: by just using raise without any argument

```
def readFile(fname):
```

```
    if fname.endswith('.txt')
```

```
        fh = open(fname)
```

```
        return fh.readlines()
```

```
    else:
```

```
        raise ValueError("filename must end with txt")
```




File Handling

File I/O

- File is a named location on disk to store related information.
- Stores data permanently store data in a non-volatile memory (e.g. hard disk).
- When we want to read from or write to a file we need to open it first.
- When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order.
 - Open a file
 - Read or write (perform operation)
 - Close the file

Opening a File

- `open()` to open a file.
- Returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
f = open("test.txt") # open file in current directory
```

```
f = open("C:/Python33/README.txt") # specifying full path
```

- We can specify the mode while opening a file.
- The default is reading in text mode

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

Closing a File

- Python has a garbage collector to clean up unreferenced objects.
- Closing a file will free up the resources that were tied with the file and is done using the close() method.

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

- If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.
- The best way to do this is using the with statement.
- This ensures that the file is closed when the block inside with is exited.
- We don't need to explicitly call the close() method. It is done internally.

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

Writing to a file

- In order to write into a file we need to open it in write 'w', append 'a' or exclusive creation 'x' mode.
- We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.
- Writing a string or sequence of bytes (for binary files) is done using write() method.
- This method returns the number of characters written to the file.

with open("test.txt",'w',encoding = 'utf-8') as f:

```
f.write("my first file\n")
```

```
f.write("This file\n\n")
```

```
f.write("contains three lines\n")
```

- This program will create a new file named 'test.txt' if it does not exist. I

Reading from a file

- To read the content of a file, we must open the file in reading mode.
- We can use the read(size) method to read in *size* number of data.
- If *size* parameter is not specified, it reads and returns up to the end of the file.

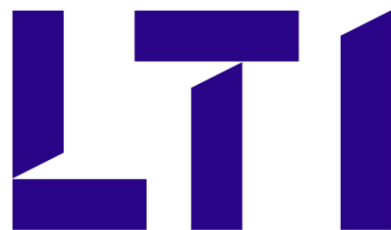
```
f = open("test.txt",'r',encoding = 'utf-8')
```

```
f.read(4) # read the first 4 data
```

- read() method returns newline as '\n'.
- Once the end of file is reached, we get empty string on further reading.
- We can change our current file cursor (position) using the seek() method.

File Methods

Method	Description
<code>close()</code>	Close an open file. It has no effect if the file is already closed.
<code>detach()</code>	Separate the underlying binary buffer from the TextIOBase and return it.
<code>fileno()</code>	Return an integer number (file descriptor) of the file.
<code>flush()</code>	Flush the write buffer of the file stream.
<code>isatty()</code>	Return True if the file stream is interactive.
<code>read(<i>n</i>)</code>	Read atmost <i>n</i> characters form the file. Reads till end of file if it is negative or None.
<code>readable()</code>	Returns True if the file stream can be read from.
<code>readline(<i>n</i>=-1)</code>	Read and return one line from the file. Reads in at most <i>n</i> bytes if specified.
<code>readlines(<i>n</i>=-1)</code>	Read and return a list of lines from the file. Reads in at most <i>n</i> bytes/characters if specified.
<code>seek(<i>offset</i>,<i>from</i>=SEEK_SET)</code>	Change the file position to <i>offset</i> bytes, in reference to <i>from</i> (start, current, end).
<code>seekable()</code>	Returns True if the file stream supports random access.
<code>tell()</code>	Returns the current file location.
<code>truncate(<i>size</i>=None)</code>	Resize the file stream to <i>size</i> bytes. If <i>size</i> is not specified, resize to current location.
<code>writable()</code>	Returns True if the file stream can be written to.
<code>write(<i>s</i>)</code>	Write string <i>s</i> to the file and return the number of characters written.
<code>writelines(<i>lines</i>)</code>	Write a list of <i>lines</i> to the file.



Let's Solve