Introduction to Python Programming

By

Mr. Ganesh Bhosale

Dwij IT Solutions,Pune

www.dwijitsolutions.com

What is Python ?

- Python is a widely used general-purpose, high-level programming language that lets you work more quickly and integrate your systems more effectively.

- Its design philosophy emphasises **code readability**, and its syntax allows programmers to express concepts in **fewer lines of code**

History of Python

- Python was conceived in the late 1980's and its implementation was started in December 1989 by **Guido van Rossum** in the Netherlands

Features of Python Language

- Very clear, readable **syntax**
- **Object orientation**
- **Natural expression** of procedural code
- Full **modularity**, supporting hierarchical packages
- Exception-based **error handling**
- Very high level dynamic **data types**
- Extensive standard libraries and third party modules for virtually every task
- Extensions and modules easily written in C, C++ (or Java for Jython)
- Embeddable within applications as a scripting interface

Core Philosophy

- **Beautiful** is better than ugly
- **Explicit** is better than implicit
- **Simple** is better than complex
- **Complex** is better than complicated
- **Readability** counts

Python facts !!!

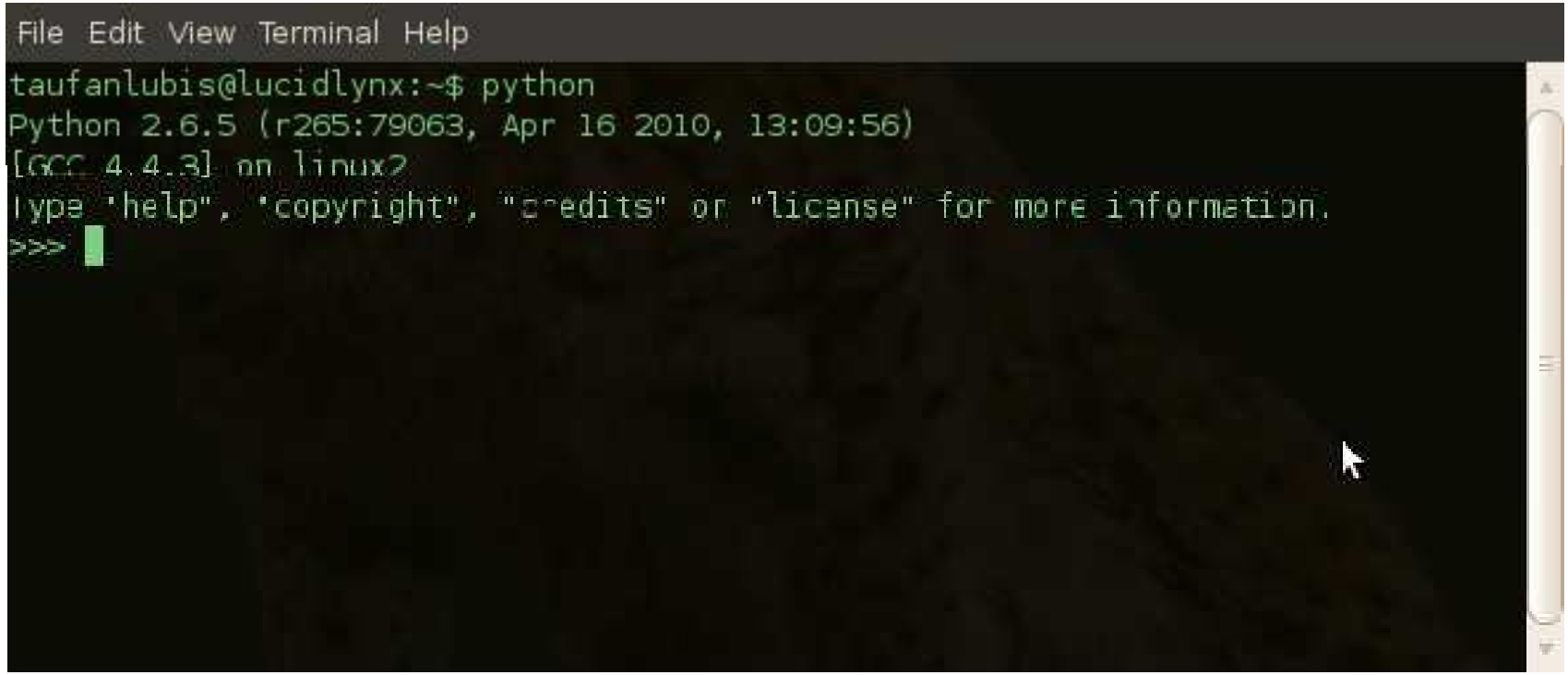
- Open Source
- **Easy** to Learn & excel.
- No need of **compilation** of code.
- **Error** handling in runtime.
- Take very less **time** for up & running.
- Object Oriented Programming + Procedural + Functional
- Pre-installed with Linux, Unix & Mac OSX

Relocating Python

- Change path
- `set path=%path%;C:\python27`

Up & running with Python

Python Interpreter



Using Python Interpreter

```
>>>(7+2)*10
90

>>>print “Hello World”
Hello World

>>>i=90
>>>print i*(i+10)
9000
```

To exit interpreter:
Use **Ctrl+D** in Linux & Unix
Use **Ctrl+Z** and then **Enter** in Windows

Running Python Files (.py)

- Off course you are not going to write all your python codes on interpreter.
- We use extension **".py"** for saving python files.
- To run python file:

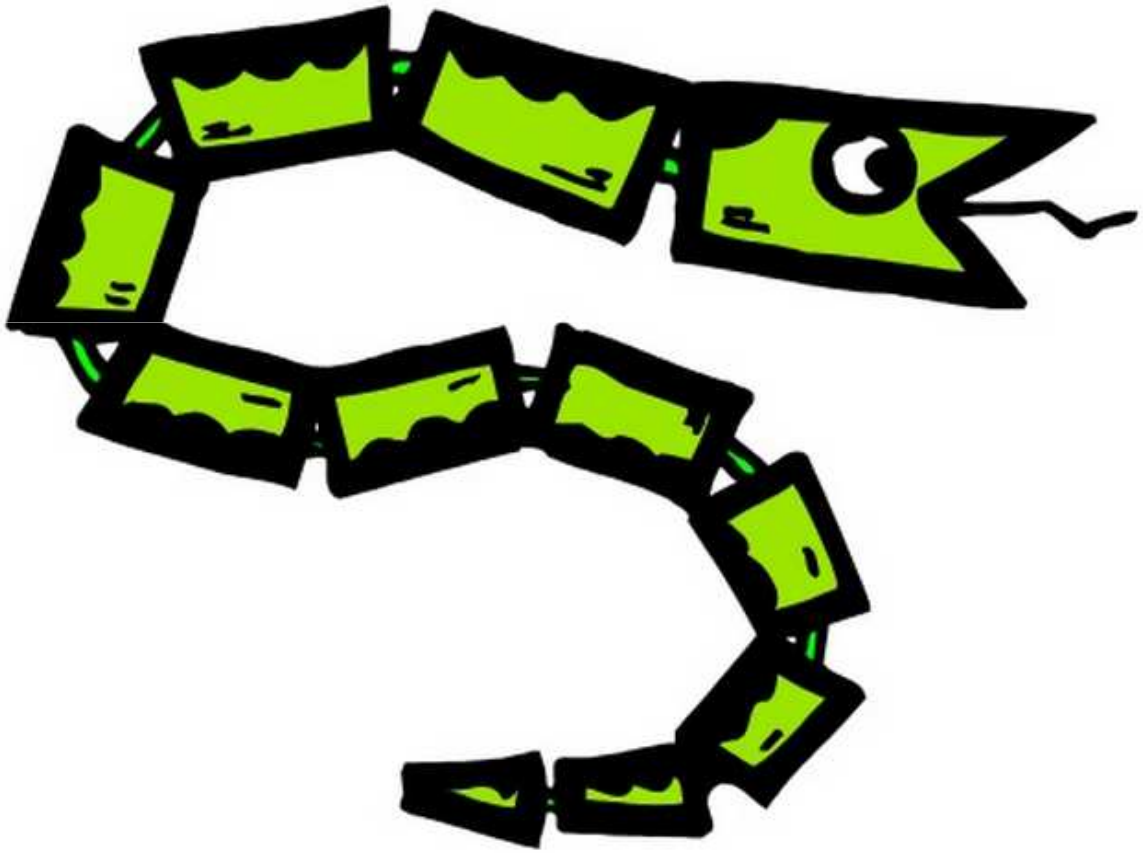
```
pi@raspberrypi ~ $python MyProgram.py
```

- Let's see How to write programs in Python !!!

SimpleProgram.py

```
x = 34 - 23        # A comment.
y = "Hello"        # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
        x = x + 1
        y = y + " World"    # String
concat.
print x
print y
```

Basics of Python



Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter **A to Z** or **a to z** or an **underscore** (_) followed by zero or more letters, underscores and digits (**0 to 9**)
- Characters not allowed: @, $ and % & other symbols

- Python is a **case sensitive**. Thus, **Manpower** and **manpower** are two different identifiers.

Coding Style

- **No braces** for block of code (We use indentation instead)
- **Indentation** with similar number of spaces / tabs.

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

This will
Generate error

Python 2 or 3?

- For beginners there is no real difference between Python 2 & 3. The basics are the same (except for print)

```
print "hello world"
```

Python 2

```
print("hello world")
```

Python 3

print is no longer a statement,
but a function in python 3

Data

Objects

Everything in *Python is an object that has:*

- an *identity (id)*
- a *type*
- a *value (mutable or immutable)*

id

```
>>> a = 4
>>> id(a)
6406896
```

type

```
>>> a = 4
>>> type(a)
<type 'int'>
```

Value

Mutable: When you alter the item, the id is still the same. Dictionary, List

Immutable: String, Integer, Tuple

Mutable

```
>>> b = []
>>> id(b)
140675605442000
>>> b.append(3)
>>> b
[3]
>>> id(b)
140675605442000 # SAME!
```

Object value can be changed.

Here object id doesn't change.

Immutable

```
>>> a = 4
>>> id(a)
6406896
>>> a = a + 1
>>> id(a)
6406872 # DIFFERENT!
```

Object value cannot be changed.

If tried, New object will be created & referred to a variable

Variables

```
a = 4            # Integer
b = 5.6          # Float
c = "hello"      # String
a = "4"          # rebound to String
```

Naming

- lowercase
- underscore_between_words
- don't start with numbers

See PEP on http://legacy.python.org/dev/peps/pep-0008

PEP

Python Enhancement Proposal (similar to JSR in Java) Java Specification Requests

- The formal documents that describe proposed specifications and technologies for adding to the platform.

- In short → Standardization

Maths

Math

+, -, *, /, ** (power), % (modulo)

Careful with integer division

```
>>> 3/4
0
>>> 3/4.
0.75
```

(In Python 3 **//** is integer division operator)

What happens when you raise 10 to the 100th?

```
>>> 10**100
10000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000L
```

LONG variable

LONG

```
>>> import sys
>>> sys.maxint
9223372036854775807
>>> sys.maxint + 1
9223372036854775808L
```

New way of defining variables

a, b = 0, 1

Is same as

a = 0

b = 1

None

Pythonic way of saying NULL. Evaluates to False.

c = None

Global Variable

```
# declare global variable
n = 0

def setup():
    global n
    n = 100
```

Booleans

```
a = True
b = False

if True:
    print "Its true"
if a:
    print a
```

First letters should be capital

Prints True

Strings

Strings

```
name = 'matt'

with_quote = "I ain't gonna"

longer = """This string has
multiple lines
in it"""
```

Multiline String

How to print complex Strings

String Escaping

Escape with \

```
>>> print 'He said, "I\'m sorry"'
He said, "I'm sorry"
>>> print '''He said, "I'm sorry"'''
He said, "I'm sorry"
>>> print """He said, "I'm sorry\"""
He said, "I'm sorry"
```

String Escaping

Escape Sequence	Output
\\	Backslash
\'	Single Quote
\"	Double quote
\b	ASCII Backspace
\n	Newline
\t	Tab
\u12af	Unicode 16 bit
\U12af89bc	Unicode 32 bit
\o84	Octal character
\xFF	Hex character

String formatting

c-like

```
>>> "%s %s" %('hello', 'world')
'hello world'
```

PEP 3101 style

```
>>> "{0} {1}".format('hello', 'world')
'hello world’
>>> print "Hello" , " " , "World"
Hello World
```

Strings manipulation

- Strings can be concatenated (glued together) with the + operator, and repeated with *

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'

>>> 'Py' 'thon'
'Python'
```

Strings manipulation

```
>>> text = ('Put several strings within '
            'to have them.')
>>> text
'Put several strings within to have them.'

>>> word[0]  # character in position 0
'P'
>>> word[-1]  # last character
'.'
```

Integer to String

- To convert Integer/Object to String

```
>>> str = str(12345)
```

String methods

s.endswith(sub)

Returns True if endswith sub

s.find(sub)

Returns index of sub or -1

s.format(*args)

Places args in string

s.index(sub)

Returns index of sub or exception

s.join(list)

Returns list items separated by string

s.strip()

Removes whitespace from start/end

Misc. Methods

dir Method

- Used to list down attributes & methods

```
>>> dir("a string")
['__add__', '__class__', ... 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- built-in function **dir()** is used to find out which names a module defines. It returns a sorted list of strings

Without arguments, dir() lists the names you have defined currently

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Dunder methods

dunder (double under) or "special/magic“

methods determine what will happen when

+ (__add__) or

/ (__div__) is called.

help

```
>>> help("a string".startswith)
Help on built-in function startswith:

startswith(...)
    S.startswith(prefix[, start[, end]]) -> bool

    Return True if S starts with the specified prefix, False otherwise.
    With optional start, test S beginning at that position.
    With optional end, stop comparing S at that position.
    prefix can also be a tuple of strings to try.

>>> str = "Hello World"
>>> str.startswith("He")
True
```

Closing help

- Shows full screen documentation
- Use page-up page-down to scroll
- Press **h** for help
- Press **q** to quit

Comments

- Comments follow a #
- No multi-line comments

- You can use # in this case*

This is multiline comment using hash
and it goes to another line as well

To take input from keyboard

- Gives input in String format

```
>>> a= raw_input()
12
>>> a
'12'
```

Print without new line

- Way 1

```
>>> for val in range(0,10):
...     print val,
0 1 2 3 4 5 6 7 8 9
```

Put Trailing comma

Print without new line

- Way 2 – This wont work without import

```
>>> from __future__ import print_function
>>>
>>> print("Hi...", end='')
Hi...>>>
```

End Character

Print without new line

- Way 3 – Creating own custom method in module

```
# printf.py

from __future__ import print_function

def printf(str, *args):
    print(str % args, end='')

# use
from printf import printf

printf('hello %s', 'world')
```

Print without new line

- Way 4 – Creating own custom module using system library

```
# print1.py
import sys
def print1(str):
   sys.stdout.write(str)
   sys.stdout.flush()

# use
from print1 import print1

print1('hello')
print1(' world')
```

Data Structures

Sequences

1. *lists*
2. *dictionaries*
3. *tuples*
4. *sets*

1. Lists

- Hold sequences.
- How would we find out the attributes & methods of a list?

```
>>> dir([])
['__add__', '__class__', '__contains__',...
'__iter__',... '__len__',... , 'append', 'count',
'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

1. Lists

```
>>> a = []
>>> a.append(4)
>>> a.append('hello')
>>> a.append(1)
>>> a.sort()        # in place
>>> print a
[1, 4, 'hello']
```

1. Lists

- How would we find out documentation for a method?

help function:

OR >>> **help**(list)

```
>>> help([].append)
Help on built-in function append:
append(...)
L.append(object) -- append object to end
```

1. List Methods

l.append(x)

Insert x at end of list

l.extend(list2)

Add list2 items to list

l.sort()

In place sort

l.reverse()

Reverse list in place

l.remove(item)

Remove first item found

l.pop()

Remove/return item at end of list

l.insert(i, x)

Insert an item at a given position

l.index(x)

Return the index in the list of the first item whose value is *x*

l.count(x)

number of times *x* appears in the list

l. sort(cmp=None, key=None, reverse=False)

Sort the items of the list in place

Using Lists as Stacks

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Nested Lists

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

2. Dictionaries

- Also called *hashmap or associative array* elsewhere

```
>>> age = {}
>>> age['george'] = 10
>>> age['fred'] = 12
>>> age['henry'] = 10
>>> print age['george']
10
>>> 'henry' in age
True
```

Find out if 'henry' in age dictionary
Uses __contains__ dunder method

.get Method

```
>>> print age[‘ford']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: ‘ford'
>>> print age.get(‘ford', 'Not found.')
Not found.
```

in keyword

- Checks if value present in sequence

```
>>> a = [10, 20, 30]
>>> 10 in a
True
>>> 25 in a
False
```

Delete Dictionary Key

Removing 'henry' from age

```
>>> del age[‘henry']
```

- del not in dir(age)
- .pop is an alternative

3. Tuples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5)                    # Tuples may be nested
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> t[0] = 88888                              # Tuples are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> v = ([1, 2, 3], [3, 2, 1])                # but they can contain mutable objects
>>> v
([1, 2, 3], [3, 2, 1])
```

3. Tuples

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)      # Check length
1
>>> singleton
('hello',)
>>> singleton = singleton + ('Aditya',)        # append to tuple
>>> singleton
('hello', 'Aditya')
>>> singleton = singleton + ('Mahesh')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "str") to tuple
```

4. Set

- Set is an unordered collection with no duplicate elements
- Also support mathematical operations like union, intersection, difference, and symmetric difference

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)          # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit            # fast membership testing
True
```

Set Operations

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                          # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                      # letters in a but not in b. relative complement
set(['r', 'd', 'b'])
>>> a | b                      # letters in either a or b. union
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                      # letters in both a and b intersection
set(['a', 'c'])
>>> a ^ b                      # letters in a or b but not both  (union- intersection)
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

In Short

Type	Defined using	Appending	Access
Lists	list = []	List.append(123)	List[0]
Dictionary	dict = {}	dict['elem'] = 123	dict['elem']
Tuples	tuple = ()	tuple = tuple + (123,)	tuple[0]
Sets	s = set(['a', 'b'])	s.add('c')	val = next(iter(a)) or a.pop()

Slicing

- Negative Indexing
- reinterpret a[-X] as a[len(a)-X]

```
colors = ["red", "blue", "purple", "maroon"]

print colors[0]      #red
print colors[-1]     #maroon
```

Negative Indexing

```
>>> a = [1, 2, 3]
>>> a[0]
1
>>> a[-1]
3
>>> a[-2]
2
>>> a[-3]
1
>>> a[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Slicing 2

- arr [start : end]

```
colors = ["red", "blue", "purple", "maroon"]

print colors[0:1]   #red
print colors[1:2]   #blue
print colors[:2]    #red, blue
print colors[2:]    #purple, maroon
```

Stride

- `arr [start : end : step]`

```
>>> a = range(0,10)[2:5:2]
>>> a
[2, 4]

>>> a = range(0,10)[::3]
>>> a
[0, 3, 6, 9]
```

Slicing 3

```
>>> veg = "ONION"
>>> veg[:-1]
'ONIO'
>>> veg[::2]
'OIN'
>>> veg[::-1]
'NOINO'
```

Control Flow

Conditionals

```
if grade > 90:
        print "A"
elif grade > 80:
        print "B"
elif grade > 70:
        print "C"
else:
        print "D"
```

Remember the colon/whitespace!

Comparison Operators

- Supports (>, >=, <, <=, ==, !=)

```
>>> 5 > 9
False
>>> 'matt' != 'fred'
True
>>> isinstance('matt', basestring)
True
```

Boolean Operators

- and, or, not (for logical), &, |, and ^ (for bitwise)

```
>>> x = 5
>>> x < -4 or x > 4
True
```

Chained comparisons

```
if 3 < x < 5:
        print "Four!"
```

Same as

```
if x > 3 and x < 5:
        print "Four!"
```

Comparing Sequences and Other Types

- **compared to other objects with the same sequence type**
- **uses *lexicographical* ordering**
- **First the first two items are compared, and if they differ this determines the outcome of the comparison**

```
(1, 2, 3)                 < (1, 2, 4)
[1, 2, 3]                 < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)              < (1, 2, 4)
(1, 2)                    < (1, 2, -1)
(1, 2, 3)                == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))      < (1, 2, ('abc', 'a'), 4)
```

range

- ([start, end)) with range and slices.
- length = end – start

```
>>> a = range(0,2)
>>> a
[0, 1]
```

Unpacking Argument Lists

- when the arguments are already in a list or tuple

```
>>> range(3, 6)
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]
```

Iteration

```
for i in range(1, 6):
    print i
```

Same as

```
for i in [1,2,3,4,5]:
    print i
```

Array Object Iteration 1

```
arr = ["str1", "str2", "str3"]

for i in range(len(arr)):
    print i, arr[i]
```

Array Object Iteration 2

```
arr = ["str1", "str2", "str3"]

for i in arr:
    print i
```

Array Object Iteration 3 enumerate

```
arr = ["str1", "str2", "str3"]

for index, value in enumerate(arr):
    print index, value
```

More looping

- entries can be paired with the zip() function

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

More Looping

```
for i in reversed(range(1,10,2)):

basket = ['apple', 'orange', 'apple', 'pear']
for f in sorted(set(basket)):

knights = {'gallahad':'the pure','robin':'the brave'}
for k, v in knights.iteritems():
  print k, v
```

continue

```
list1 = [12, 50, 34, 78, 33, 99]

for item in list1:
    if item < 50:
        continue
    print item
```

Find Even & Odd numbers

```
for num in range(2, 10):
  if num % 2 == 0:
     print num, "Even"
     continue
  print num, "Odd"
```

break

```
list1 = [12, 34, 78, 99]

for item in list1:
    print item
    if item > 70:
        break
```

Prime Number Program

- A prime or prime number is a natural number that has exactly two distinct **natural number divisors**: **1** and itself

```
for n in range(2, 10): # total range of numbers
      for x in range(2, n): # get numbers less than n
            if n % x == 0: # check modulo
                  print n, '=', x, '*', n/x
                  break
      else:
            # loop fell without finding factor
            print n, 'Prime number'
```

Process Dictionary

```
d1 = { "john": 66, "mike": 81, "rock": 77}

print "Key list-----"
for key in d1.keys():
    print key

print "Value list-----"
for value in d1.values():
    print value

print "Key Value pairs-----"
for key, value in d1.items():
    print "[",key," : ",value,"]"
```

pass

- Pass is null operation
- Used as placeholder for syntactical needs

```
for i in range(10):
    pass

def fun(arg):
    pass

class c:
    pass
```

A Fibonacci Series

```
>>> # Fibonacci series:
... # sum of two elements defines next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
```

A Fibonacci Series (Horizontal)

```
>>> # Fibonacci series:
... # sum of two elements defines next
... a, b = 0, 1
>>> while b < 10:
...     print b,
...     a, b = b, a+b
...
```

A suggestion

Don't modify list or dictionary contents while looping them.

Functions

Functions

```
def add_2(num):
    """ return 2
    more than num
    """
    return num + 2
five = add_2(3)
```

Functions

- def
- function name
- (parameters)
- : + indent
- optional documentation
- body
- return

Whitespace

- Instead of { use a : and indent consistently (4 spaces)
- invoke python -tt to error out during inconsistent tab/space usage in a file

Default (named) parameters

```
def add_n(num, n = 3):
        """default to
        adding 3"""
        return num + n
five = add_n(2)
ten = add_n(15, -5)
```

Default parameters can be mutable

- when the default is a mutable object such as a list, dictionary, or instances of most classes

```
def f(a, L=[]):
  L.append(a)
  return L

print f(1)
print f(2)
print f(3)
```

[1]
[1, 2]
[1, 2, 3]

To change to default behavior

- If you don't want the default to be shared between subsequent calls

```
def f(a, L=None):
  if L is None:
     L = []
  L.append(a)
  return L
```

[1]
[2]
[3]

__doc__

- Functions have *docstrings. Accessible* via .__doc__ or help

```
>>> def echo(txt):
...         "echo back txt"
...         return txt
>>> help(echo)
Help on function echo in module __main__:
<BLANKLINE>
echo(txt)
        echo back txt
<BLANKLINE>
```

Function/Method Documentation

Method Naming

- Lowercase
- underscore_between_words
- don't start with numbers
- verb
- See PEP 8 for more info

Function Properties

```
>>> def addition(a, b=0):
...     "This method adds two numbers"
...     return a+b
...
>>>
>>> addition(10,20)
30
>>> addition(10)
10
>>> addition
<function addition at 0x01B24B70>
>>> add = addition
>>> add(10, 30)
40
```

Function refers to another function variable

File I/O

File Input

- Open a file to read from it

```
fin = open("sample.txt")
for line in fin:
    print line
fin.close()
```

File Output

- Open a file using 'w' to write to a file: Open a file using 'w' to write to a file:

```
fout = open("sample2.txt", "w")
fout.write("hello world")
fout.flush()
fout.close()
```

Suggestion

Don’t forget to close file after use.

Closing files using **with**

- Also called as implicit close (Only ver. 2.5+)

```
with open('sample.txt') as fin:
    for line in fin:
        print line
```

Classes

Classes

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def study(self):
        print self.name,"studying..."

stud = Student("Mark Watson")
stud.study()
```

Constructor

Classes

- object (as base class)
- dunder __init__ (constructor)
- all methods take self as first parameter

self

- Always represent current object

Subclass

```
class Geek(Student):
    "classes can have documentation"
    def study(self):
        print "%s Programming..." % self.name

g = Geek("Mark Watson")
g.study()
```

Class Naming

- Cannot start with numbers
- Nouns
- **CamelCase** – first the letter in a word within name should be written in capitals

Modules

Modules

- Put definitions in a file and use them in a script or in an the interpreter. Such a file is called a ***module***;
- A module is a file containing Python definitions and statements.
- module name with the suffix .py appended

fibo.py

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Using Module

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Using Module

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

>>> import fibo as f
>>> f.fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

sleep

- Sleep function pauses the execution for given number of seconds

```
from time import sleep

while True:
    print "Hi"
    sleep(1)
```

Executing module as script

```
$ python fibo.py <arguments>
```

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Accessing arguments via system library

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

```
>>> import fibo
>>>
```

If the module is imported, the code is not run

Import library from another directory

- Way 1 – Not working

```
pi@raspberrypi ~/practice2 $ export PATH=$PATH:/home/pi/practice2/lib.printf.py
```

Import library from another directory

- Way 2
- http://stackoverflow.com/questions/279237/import-a-module-from-a-relative-path

```
import sys
sys.path.append(“/home/pi/prac1/Student”)
from Student import Student
s = Student("Ganesh")
```

Compile Library without execution

- python -m py_compile printf.py