Day1

Lession1

- **Introduction to python**
  - Created in 1990 by Guido van Rossum
    - While at CWI, Amsterdam
    - Now hosted by centre for national research initiatives, Reston, VA, USA
  - Free, open source
    - And with an amazing community
  - Object oriented language
    - "Everything is an object"

- **Why python?**
  - Simple & Elegant & Fast enough for processing large files
  - Language for both beginners & experts
  - Supports OOPs (Object Oriented Programming)
  - Cross platform
  - Easy to learn
  - Easy to deploy
  - Easy threading & thread management
  - Efficient – works well with big data technologies as well
  - Its Open Source, hence many different python(s) eg PyPy, jython, PyObjc, ironPython etc. That's means it's extensible
  - Used in almost all the fields for faster computation – there is less IO, hence fast
- **Python environment**

  Installer: python-3.4.3.msi

  If we have multiple version of python installed, we can run our prompt like below

  Running with multiple version

  Cmd->Py -<version> <filename>.py

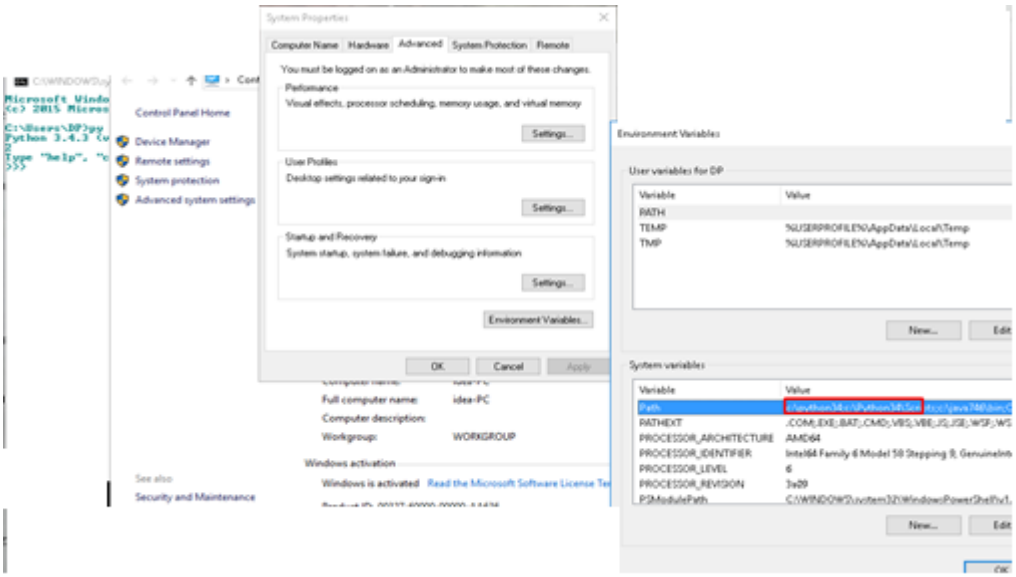
  Py -2.7 abc.py, Py -3.4 abc.py, Py -3.5 abc.py,

  Shell:

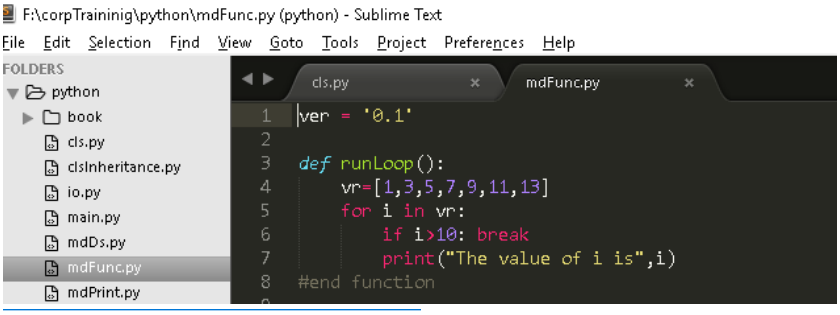
```
:\Users\DP>py
ython 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win3

ype "help", "copyright", "credits" or "license" for more information.
>>>
```

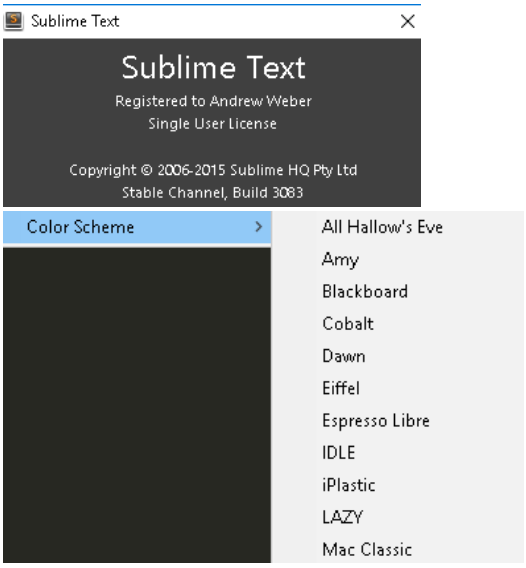
```
F:\corpTraininig\python>py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on
2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

  Environment variable



- **Familiarising with UI – sublime-text**





Lession2

- **Strings, date time, console output**

```
F:\corpTraininig\python>stringDateTimeExample.py
2016-09-06 13:33:00
Updated str1ing :- Hello Python
My name is Zara and weight is 21 kg!
this is a long str1ing that is made up of
several lines and non-printable characters such as
TAB (	) and they will show up that way when displayed.
NEWLINEs within the str1ing, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.

str1.count(sub, 4, 40) :  2
str1.count(sub) :  1
16
16
16
16
-1
False
True
a-b-c
THIS IS STR1ING EXAMPLE....WOW!!!
this is STR1ing example....wow!!!
total characters in str1ing: 33
0000000this is str1ing example....wow!!!
0000000000000000000this is str1ing example....wow!!!
string repeat using *: **********
using zfill: 004
004
004
004
004
004
004
```

stringDateTimeExam
ple.py

- **Control flows – if, for, range, break, continue, pass**

```
F:\corpTraininig\python>controlFlow.py
Negative
100
200
300
0
1
2
3
4
5
4
3
2
1
0
2
4
1 1
2 4
3 9
1 a
2 b
Index of dog: 1
Greater than 4: 5
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
```


controlFlow.py

Lession3

- **Defining functions**

```
F:\corpTraininig\python>funcCheck.py
I'm first call to user defined function!
Again second call to the same function
```


funcCheck.py

- **Default argument values**

```
F:\corpTraininig\python>defaultCheck.py
Example1: the static list length is: 1
Example1: the static list length is: 2
Example1: the static list length is: 1
Example1: the static list length is: 3
Example2: the static list length is: 1
Example2: the static list length is: 2
Example2: the static list length is: 1
Example2: the static list length is: 3
LOG: Print my message
```


defaultCheck.py

- **Keyword arguments**

On the calling side, you have the ability to specify some function arguments by name. You have to mention them after all of the arguments without names (positional arguments), and there must be default values for any parameters which were not mentioned at all.

```
def my_function(arg1, arg2, **kwargs)
```

The other concept is on the function definition side: You can define a function that takes parameters by name -- and you don't even have to specify what those names are. These are pure keyword arguments, and can't be passed positional. The syntax is
see attached code as well

```
F:\corpTraininig\python>keyArgs.py
{'b': 'abc', 'a': 12}
Positional:  ('one', 'two', 'three')
Keywords:  {}
Positional:  ()
Keywords:  {'c': 'three', 'b': 'two', 'a': 'one'}
Positional:  ('one', 'two')
Keywords:  {'c': 'three', 'd': 'four'}
a:a, b:b, c:c
a:z, b:q, c:v
```


keyArgs.py

- **Arbitrary argument list**

  Variable number of arguments

```
F:\corpTraininig\python>arbitArgs.py
I was called with 1 arguments: (1,)
I was called with 3 arguments: (1, 2, 3)
(5, 2, 'omega')
{'key1': 5, 'key2': 7}
```


arbitArgs.py

- **Unpacking argument list**

  The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

- **Lambda expressions**

  When function objects are required or pass function as argument

```
F:\corpTraininig\python>lambda.py
The Sum by function object is: 42
The Sum by function object is: 87
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```


lambda.py

- **Documentation strings**

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

- **Function annotations(optional)**

  Strong type check

  Return values

```
def foo(a, b: 'annotating b', c: int) -> float:
    print(a + b + c)
```

```
F:\corpTraininig\python>functionAnnotation.py
Hello 1, you are 28 years old
Hello Devendra Prasad, you are 32 years old
```


functionAnnotation.py

Lession4

- **Data Structures**
  - **More on lists**
    - **Using list as stacks**

```
F:\corpTraininig\python>listAsStack.py
Stack by class example
True
dog
3
False
8.4
True
2
Exampe of list as stack
[1, 2, 3, 4, 5, 6, 7, 8, 9]
Delete value is: 9
[1, 2, 3, 4, 5, 6, 7, 8, 20]
```


listAsStack.py

- **Using list as queues**

```
F:\corpTraininig\python>listAsQ.py
deque(['rohan', 'sameer', 'adil', 'saksham'])
deque(['rohan', 'sameer', 'adil', 'saksham', 'priya', 'aashi'])
deque(['sameer', 'adil', 'saksham', 'priya', 'aashi'])
```


listAsQ.py

- **List comprehensions(works on py 3 onwards)**

  Command can be: py -3 listComprehension.py

```
F:\corpTraininig\python>c:\Python34\python.exe listComprehension.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
[0, 4, 16, 36, 64]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 6
9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 8, 12, 16, 20, 24, 28, 32, 36, 40, 4
 48, 10, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 21, 28, 35, 42, 49]
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
['THE', 'the', 3]
['QUICK', 'quick', 5]
['BROWN', 'brown', 5]
['FOX', 'fox', 3]
['JUMPS', 'jumps', 5]
['OVER', 'over', 4]
['THE', 'the', 3]
['LAZY', 'lazy', 4]
['DOG', 'dog', 3]
```


listComprehension.py

Few more examples

print [i for i in range(10)]

print [i for i in range(20) if i%2 == 0]

print [(i,f) for i in nums for f in fruit]

print [(i,f) for i in nums for f in fruit if f[0] == "P"]

print [(i,f) for i in nums for f in fruit if f[0] == "P" if i%2 == 1]

print [i for i in zip(nums,fruit) if i[0]%2==0]

- **Nested list comprehensions (3onwards)**

```
F:\corpTraininig\python>listNested.py
['40', '20', '10', '30']
['20', '20', '20', '20', '20', '30', '20']
['30', '20', '30', '50', '10', '30', '20', '20', '20']
['100', '100']
['100', '100', '100', '100', '100']
['100', '100', '100', '100']
[[40.0, 20.0, 10.0, 30.0], [20.0, 20.0, 20.0, 20.0, 20.0, 30.0, 20.0], [30.0, 20.0, 30.0,
0, 10.0, 30.0, 20.0, 20.0, 20.0], [100.0, 100.0], [100.0, 100.0, 100.0, 100.0, 100.0], [10
, 100.0, 100.0, 100.0]]
```


listNested.py

- **The del statement**

  Python also provides a mechanism for removing variables / memory cleanup, the **del** statement.

  Del 〈 ***object*** **,...**〉

```
del list_item[4]
del dictionary["alpha"]
```

```
del foo
```

One place I've found `del` useful is cleaning up extraneous variables in for loops:

```
for x in some_list:
  do(x)
del x
```

- **Tuples and sequences**

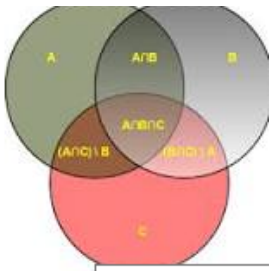
  A tuple is a sequence of immutable(not changeable) Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed. unlike lists and tuples use parentheses, whereas lists use square brackets.
  Creating a tuple is as simple as putting different comma-separated values.

```
F:\corpTraininig\python>tupleSeq.py
('tup1[0]: ', 'physics')
('tup2[1:5]: ', (2, 3, 4, 5))
entire tuple object: (12, 34.56, 'abc', 'xyz')
element at 4th position: xyz

F:\corpTraininig\python>py -3 tupleSeq.py
tup1[0]:  physics
tup2[1:5]:  (2, 3, 4, 5)
entire tuple object: (12, 34.56, 'abc', 'xyz')
element at 4th position: xyz
```


tupleSeq.py

- **Sets**

The data type "**set**", which is a collection type, has been part of **Python** since version 2.4. A **set** contains an unordered collection of unique and immutable objects. The **set** data type is, as the name implies, a **Python** implementation of the **sets** as they are known from mathematics.



```
F:\corpTraininig\python>setCheck.py
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
is engineer a superset in employee: False
is engineer a superset in employee: True
Set(['Jane', 'Marvin', 'Janice', 'John', 'Jack'])
Set(['Janice', 'Jack', 'Sam'])
Set(['Jane', 'Zack', 'Jack'])
Set(['Zack', 'Sam', 'Marvin', 'Jack', 'Jane', 'Janice', 'John'])
```


setCheck.py

- **Dictionaries**

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

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

```
F:\corpTraininig\python>dictCheck.py
dict['Name']:  Zara
dict['Age']:  7
----------
NY State has:  New York
OR State has:  Portland
----------
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
----------
Michigan has:  Detroit
Florida has:  Jacksonville
----------
California is abbreviated CA
Michigan is abbreviated MI
New York is abbreviated NY
Florida is abbreviated FL
Oregon is abbreviated OR
----------
FL has the city Jacksonville
CA has the city San Francisco
MI has the city Detroit
OR has the city Portland
NY has the city New York
----------
California state is abbreviated CA and has city San Francisco
Michigan state is abbreviated MI and has city Detroit
New York state is abbreviated NY and has city New York
Florida state is abbreviated FL and has city Jacksonville
Oregon state is abbreviated OR and has city Portland
----------
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist
```

dictCheck.py

- **More on conditions**

```
if (cond1 == 'val1' and cond2 == 'val2' and
       cond3 == 'val3' and cond4 == 'val4'):
    do_something
```

```
if (
       cond1 == 'val1' and cond2 == 'val2' and
       cond3 == 'val3' and cond4 == 'val4'
   ):
    do_something
if    (cond1 == 'val1' and cond2 == 'val2' and
       cond3 == 'val3' and cond4 == 'val4'):
    do_something
```

```
if cond1 == 'val1' and cond2 == 'val2' and \
                       cond3 == 'val3' and \
                       cond4 == 'val4':
    do_something
```

```
if cond1 == 'val1' and \
   cond2 == 'val2' and \
   cond3 == 'val3' and \
   cond4 == 'val4':
    do_something
```

```
if all( [cond1 == 'val1', cond2 == 'val2', cond3 == 'val3', cond4 == 'val4'] ):

if any( [cond1 == 'val1', cond2 == 'val2', cond3 == 'val3', cond4 == 'val4'] ):
```

Ref: http://stackoverflow.com/questions/181530/python-style-multiple-line-conditions-in-ifs

- **Comparing sequence and other types**

```
F:\corpTraininig\python>diffsObj.py
{'a': 2, 'c': 3, 'b': 2, 'd': 4} {'c': 3, 'b': 2, 'd': 4}
UnMatched Items: set([('a', 2)])
Matched Items: set([('b', 2), ('c', 3), ('d', 4)])
Loop over matched items and print elements
('b', 2)
('c', 3)
('d', 4)
```

Day2

Lession5

- **Modules**

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

Follow fibo.py module and call it in some other python file like as below

```
import fibo
# from fibo import fib, fib2
fibo.fib(10)
```

6.1.1. Executing modules as scripts

When you run a Python module with

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

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

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

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

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

- **Standard modules**

6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Find out the list of all standard modules?

- **DIR() function**

Consider above **fibo.py** example and use of dir() will give complete list of vars and functions available in there

```
1 1 2 3 5 8
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'fib', 'fib2']
F:\corpTrainingig\python\main.py
```

```
from os import path

dirname = path.dirname(__file__)
if dirname == "":
    dirname = "."
```

- **Packages**

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
```

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

- **.pyc file – compiled moduel**
  Whenever you import python script into another python script, the called script become the .pyc file i.e the compiled module

Lession6

- **Input and output**
  read from console: int(input('Enter an integer : '))
  write to console: print or print(“”)
- **Reading and writing files**

```
f = open(fullPath)
while True: #loop invite and read line by line and exit when there is nothing to read i.e line length = 0
    line = f.readline()
    if len(line) == 0:
        break
    print(line)
f.close
```

```
#mode: r, w, a, b
poem = "Programming is fun\nWhen the work is done\nif you wanna make your work also fun"
f = open(fullPath, w)
f.write(poem)
f.close()
```

Lession7

- **Error and exceptions**
  - **Syntax errors**

8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

  - **Exceptions**

8.2. Exceptions

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 a called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by program however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

  - **Handling & Raising exceptions**

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

```
... except (RuntimeError, TypeError, NameError):
```

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
```

```
try:
    print(2/0)
except Exception as e:
    print(e)
```

  - **User defined exceptions**

```
F:\corp\rainingg\python>userDefinedError.py
multiply operation: 8
deivide operation: 5
user defined exception has occured:divide by 0 has come
```


userDefinedError.py

  - **Defining clean-up actions**

use of finally block to do any cleanup activities

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
```

- **Predefined clean-up actions**

  Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print line,
```

File handle will be open for infinite time and could be an issue in large processing.

use of with keyword resolves the problem by auto clean up or sort of garbage collection

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

Lession8

- **Classes (objects, instances, methods)**

Classes give us the ability to create more complicated data structures that contain arbitrary content. We can create a `Pet` class that keeps track of the name and species of the pet in usefully named attributes called `name` and `species`, respectively.

```
F:\corpTraininig\python>cls.py
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.
Robots can do some work here.
Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```


cls.py

- **Inheritance**

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

```
class DerivedClass(BaseClass):

    body_of_derived_class
```

```
F:\corpTraininig\python>py -3 clsInheritance.py
Initialise school member:  Vishal
Initialise school member:  Devendra
Name: Vishal Age: 36
Salary: 352353
Name: Devendra Age: 31
Marks: 224
```


clsInheritance.py

```
F:\corpTraininig\python\inheritenceTest>main.py
Enter side 1 : 3
Enter side 2 : 4
Enter side 3 : 5
('Side', 1, 'is', 3.0)
('Side', 2, 'is', 4.0)
('Side', 3, 'is', 5.0)
The area of the triangle is 6.00
```


inheritenceTest.zip

- **Iterators**

```
for i in [1, 2, 3, 4]:
    print i,
for c in "python":
    print c
for k in {"x": 1, "y": 2}:
    print k
for line in open("a.txt"):
    print line,
```

So there are many types of objects which can be used with a for loop. These are called iterable objects.

There are many functions which consume these iterables.

```
",".join(["a", "b", "c"])
› ",".join({"x": 1, "y": 2})
list("python")
list({"x": 1, "y": 2})
```

Iterator

```
x = iter([1, 2, 3])
```

x is an iterator with the user **iter** keyword, we can get an iterator object

x.next() – gets the next member from iterator

reverse iterator

```
it = reverse_iter([1, 2, 3, 4])
```

it.next() – gets the next member from iterator in reverse order

- **Generators**

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Each time the `yield` statement is executed the function generates a new value.

generator.py

- **Generator expressions**

Generator expressions as a high performance, memory efficient generalization of list comprehensions and generators

 # Generator expression

(x*2 for x in range(256))

 # List comprehension

[x*2 for x in range(256)]

list comprehensions are better when you want to iterate over something multiple times

Basically, use a generator expression if all you're doing is iterating once. If you want to store and use the generated results, then you're probably better off with a list comprehension

Since performance is the most common reason to choose one over the other, my advice is to not worry about it and just pick one; if you find that your program is running too slowly, then and only then should you go back and worry about tuning your code

- **Scope and namespaces**

Namespace

Everything in Python is an object

For example, when we do the assignment a = 2, here 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, id() i.e. a name got an space in memory

```
>>> a = 2
>>> id(2)
507098816
>>> id(a)
507098816
```

Scope

Consider below example

```
def outer_function():
    b = 20
    def inner_func():
        c = 30

a = 10
```

At any given time there can be any of three or all three instances

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

   Consider below example

```
def outer_function():
    b = 20
    def inner_func():
        c = 30

a = 10
```

Here, the variable a is in the global namespace. Variable b is in the local namespace ofouter_function() and c is in the nested local namespace of inner_function(). When we are ininner_function(), c is local to us, b is nonlocal and a is global. We can read as well as assign new values to c but can only read b and c from inner_function(). If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b. Same thing happens when we assign a value to a.

- **Pass keyword**

  Python pass Statement. Advertisements. It is used when a statement is required syntactically but you do not want any command or code to execute. The **pass** statement is a null operation; nothing happens when it executes.

Lession9

- **Multi-threading**

```
F:\corpTraininig\python>py -3 simpleThread.py
worker thread starting
worker_1 thread starting
worker thread exiting
worker_1 thread exiting

F:\corpTraininig\python>simpleThread.py
worker thread startingworker_1 thread starting


worker thread exiting
worker_1 thread exiting
```

- **Templating**

  It's a vast logic / concept / design pattern which is primarily used in web frameworks to generate automatic code as per pattern needed. Good to know web templating framework

  Jinja2, Django templates, mako, string templates

  At run time, templates variables are replaced with the values being supplied into them either from web service (SOAP), REST, or from Database (JSON Doc) or python dictionary object

  Check below references for more details around the templating work

- **Week references**

  A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

Weak reference objects have no methods and no attributes besides `ref.__callback__`. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns `None`

```
>>> del o, o2
>>> print(r())
None
```

Lession10

- **Connecting to Oracle Database**
  - **Download respective database driver compatible to your python version e.g. in our case we installed cx_Oracle-5.2.1-11g.win32-py3.4**
  - **Its gets loaded in lib\site-packages**
  - **Open a new python program**
    - **Import the driver**
    - **Open connection by passing either DSN or connection parameter in the connecting constructor or methods exposed**
    - **Pass sql query in desired format**
    - **Execute query and get resultset back**
    - **Results may differ in their return type depending upon the database drivers used. E.g. in oracle TUPLE is by default which is immutable in nature. We can convert it in any format like dictionary or list as per need**
    - **You can loop over the resultset to expose data to file or do anything with that data**
    - **Close connection**
    - **See example attached**

oracle.py

Build python console application – exe application

http://www.py2exe.org/index.cgi/Tutorial

blog on straight win32 dll

https://bytes.com/topic/python/answers/43298-making-dll-python

Some useful references

https://automatetheboringstuff.com/

https://docs.python.org/2/library/functions.html

http://effbot.org/zone/default-values.htm

https://docs.python.org/2/tutorial/controlflow.html#arbitrary-argument-lists

https://docs.python.org/2/tutorial/modules.html

http://www.py2exe.org/old/

https://docs.python.org/2/tutorial/errors.html

https://wiki.python.org/moin/Templating

https://www.fullstackpython.com/template-engines.html

https://docs.python.org/2/library/weakref.html

https://docs.python.org/3/library/weakref.html

http://anandology.com/python-practice-book/iterators.html

https://www.python.org/dev/peps/pep-0289/

https://www.python.org/dev/peps/pep-0202/