Unit 4
Python Essentials

(The 'must-know' things about Python programming)

1. The 'lambda' operator/function

'lambda' operator or lambda function is a way to create small anonymous functions (functions without names!)

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 filter(), map() and reduce().

Syntax:

var = lambda argument_list : expression

eg(1):

```
>>>f = lambda x, y, z : x + (y * z)
>>>f (10, 4, 5)
30
```

eg(2):

```
>>> bigger = lambda x, y: x if x>y else y
>>>bigger(10, 2)
10
>>>bigger(10, 12)
12
```

eg(3):

```
>>>L = [lambda x:x**2, lambda x:x**3, lambda x:x**4]
>>>for f in L:
. . .       print(f(2))
. . .
4
8
16
```

When to use lambda?
1. the function is fairly simple
2. it is going to be used only once

2. The map function

The map() maps a function definition to a sequence.
The advantage of lambda() can be seen when it is used in combination with the map() function with two arguments:

Syntax: *r = map(func, seq)*

eg:
Consider the temperature conversion function..

Using conventional function definition,

```
>>>def farenheit(T):
...     return ( float(9) / 5) * T + 32
...

>>>t1 = 36.5, 37, 37.5, 39
>>>F = map(farenheit, t1)

>>>for ele in F:
...     print (ele)
...
# prints the contents of F
```

2(a). Using lambda() and map() together!

Consider the above example of temperature conversion, by using lambda() we need not have to define or name the function..

```
>>>F = map(lambda  x: (float(9)/5) * x + 32, t1)

>>>for ele in F:
...     print (ele)
...
# prints the contents of F
```

3. The filter function

The function offers an elegant way to filter out all the elements of a list, for which the function returns True.

Syntax:

```
res = filter(func, list)

# func - a Boolean function,
  list - this 'func' will be applied to every element of the list,
 res - list of element(from 'list') for which func returns a True value
```

eg (1):

```
>>>fib = [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
>>> result = filter(lambda x : x % 2,  fib)
>>> print result
[ 1, 1, 3, 5, 13, 21, 55 ]

#ver 3.x
>>>for ele in result: print(ele)
```

eg (2):

```
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result
[0, 2, 8, 34]
```

4. The reduce function

Syntax:
 # In Python 2.x
reduce(func, seq)

#In Python 3.x
import functools
functools.reduce(func, seq)

The function *reduce(func, seq)* continually applies the function func() to the sequence seq. It returns a single value.

If seq = [s1, s2, s3, ... , sn], then calling *reduce(func, seq)* works like this:

(a) At first the first two elements of seq will be applied to func,
i.e. *func(s1,s2)*.
The list on which reduce() works looks now like this:
[func(s1, s2), s3, ... , sn]

(b) In the next step func will be applied on the previous result and the third element of the list, i.e. *func(func(s1, s2),s3)*
The list looks like this now: *[func(func(s1, s2),s3), ... , sn]*

(c) Continue like this until just one element is left and return this element as the result of reduce()

eg(1):
>>>reduce(lambda x, y : x + y, fib)
143

eg(2):
 # what does this function print??

>>>l1 = [4, 6, 112, 76, 89, 34, 22, 10, 9, 40]

>>>reduce(lambda x, y : x if (x > y) else y, l1)
?

5. Importing user-defined modules

Step 1: Define the function in a script, save it with '.py' extension.
Do not call the function in the script!
Step 2: Open a new script and use import statement to access the function

eg: In 'pac.py'
```
import newt
n=int(input('Enter a number:'))
print('Number \t Square root\t')
for i in range(1,n):
   print( i, '\t', newt.newton(i))
```

In 'newt.py'
```
def newton(a):
   x=a
```

```
    while True:
        y = (x+a/x)/2
        if y == x:
            break
        x = y
    return y
```

6. Command-line arguments

Python accepts arguments/inputs thru command line with the use of 'sys' package.

eg: If the below python script is saved as 'test1.py'

```
import sys
print('The parameters are:' sys.argv)
print('No. of parameters: ',len(sys.argv))

#executing this script at the command prompt..

$ python test1.py
The parameters are: [ 'test1.py' ]
No. of parameters: 1
$

$ python test1.py 1 2 3 4
The parameters are: [ 'test1.py', '1', '2', '3', '4' ]
No. of parameters: 5
$
```

Note: The list of arguments in sys.argv will be inherently of type 'str'.
In case, you need to perform any arithmetic operations, they need to be converted to 'int' of 'float'.

Exercise:
Write a python program that takes a list of numbers from the command prompt and displays its sum, average, max and min values.

7. Gather Positional arguments with '*'

Recall the use of '*' in gather concept of tuples..

'*' is not a pointer operator in Python!

When used inside the function with a parameter, an asterisk groups a variable number of positional arguments into a tuple of parameter values.

eg:

```
>>>def print_all(*arg):
...       print('The positional argument tuple : ',arg)
...
>>>
>>>print_all(3.4, 55 , 'Hi!')
The positional argument tuple : ( 3.4, 55, 'Hi!' )
```

8. Gather Keyword arguments with **

By using '**' operator, keyword arguments are grouped into a dictionary, where the argument names are the keys and their values are the corresponding dictionary values.

eg:

```
>>>def prin(**kar):
. . .       print(kar)
. . .

>>>prin(branch='CSE', semester = 'Fifth', college='NIE')
{'branch' : 'CSE', 'semester' : 'Fifth', 'college' : 'NIE' }
```

9. Functions are 'First-Class' Citizens

In Python everything is an 'object'! This includes numbers, strings, tuples, lists, dictionaries and functions, as well.

Functions:

- can have variables assigned to them,
- can use them as arguments to other functions
- return them from other functions

Consider this example:

eg(1):

```
>>>def answer():
. . .       print(50)
. . .
>>>
>>>answer()
50
```

Now, let's define another function, which has one argument called 'func', a function to run.

```
>>>def run_this(func):
. . .       func()
. . .
>>>
```

Let's pass the previous function 'answer' for the above function:

```
>>>run_this(answer)
50
```

Here, the function 'answer' is used as a data!

eg(2):

Let's define a function with arguments..

```
>>>def add_args(a1,a2):
. . .       print(a1+a2)
. . .
>>>
```

Now, let's make this function as an argument to another function..

```
>>>def run_with_args(func, ar1, ar2):
...     func(ar1, ar2)
...
>>>

>>>run_with_args(add_args, 10, 25)
35
```

10. Generators

Iterable: When a list is created, the elements in the list can be read and printed using a loop..

```
>>>mylist = [ 1, 2, 3 ]
>>>for ele in mylist:
...     print(ele)
...
1
2
3
```

'mylist' is an iterable. We can use comprehension to create a list and display..

```
>>>mylist = [ x * x  for x in range( 1, 4 ) ]
>>>for ele in mylist:
...     print(ele)
...
1
4
9
```

A Generator - a Python sequence creation object.
You can iterate thru potentially huge sequences, at once, without creating and storing the entire sequence in memory.

Hence, Generators are iterators, but can be iterated over only once!!
As they do not store values in memory...they generate them on fly

The simplest generator we have known in Python is 'range()'.

Creating our own generators...

```
>>>mygen = (x  for x in range(1,6))
>>>for ele in mygen:
...     print(ele)
...
1
2
3
4
5
>>>
>>>for ele in mygen:
...     print(ele)
...
```

```
>>>
```

Usage of ***'yield'*** keyword:
'yield' is a keyword that is used instead of 'return' in a function, except that the function will return a generator

```
>>>def create_generator():
...     mylist = range(1,4)
...     for i in mylist:
...         yield i*i
...
>>>
>>>mygenerator = create_generator()
>>>for i in mygenerator:
...     print(i)
...
1
4
9
>>>
```

12. Decorators

Sometimes, you want to modify an existing function without changing its source code.
Eg: To add a debugging statement to see what arguments were passed in.

A 'decorator' is a function that takes one function as input and returns another function.

eg(1):

The function 'document_it()' defines a decorator that will do the following:
- Print the function's name with arguments
- Run the function with the arguments
- Print the result
- Return the modified function for use

```
>>>def document_it(func):
...     def new_func(*args, **kargs):
...         print('Running function :', func._ _name_ _)
...         print('Positional arguments :', args)
...         print('Keyword arguments :', kargs)
...         result = func(*args, **kargs)
...         print('Result :', result)
...         return result
...     return new_func
...
>>>
```

Using the above code...

```
>>>def add_nums(a,b):
...     return a+b
...
>>>add_nums(3, 4)
7
```

Option 1:

```
>>>cool_add_nums = document_it(add_nums)
>>>cool_add_nums(3, 4)
Running function : add_nums
Positional arguments :(3, 4)
Keyword arguments : {}
Result : 7
7
```

Option 2:

Using '*@decorator_name*'
Just add this before the function you want to decorate!!

```
>>>@document_it
. . .  def add_nums(a,b):
. . .       return a+b
. . .


>>>add_nums(3, 4)
Running function : add_nums
Positional arguments :(3, 4)
Keyword arguments : {}
Result : 7
7
```

eg(2):
Adding more than one decorator to a function..

```
>>>def square_it(func):
. . .       def new_func(*args, **kargs):
. . .           result = func(*args, **kargs)
. . .           return result*result
. . .       return new_func
. . .
>>>

>>>@document_it
. . . @square_it
. . . def add_nums(x, y):
. . .       return x+y
. . .
>>>
>>>add_nums(3, 4)
Running function : add_nums
Positional arguments :(3, 4)
Keyword arguments : {}
Result : 49
49

>>>@square_it
. . . @document_it
. . . def add_nums(x,y):
. . .       return x+y
. . .
```

```
>>>
>>>add_nums(3, 4)
Running function : add_nums
Positional arguments :(3, 4)
Keyword arguments : {}
Result : 7
49
```

A real-world example:

For calculating time complexity of a function, a wrapper function can be used (decorator function).

```
>>>import time
>>>def timing_fn(func):
...         '''
...         outputs the time a function takes to execute
...         '''
...         def wrapper():
...             t1=time.time()
...             func()
...             t2=time.time()
...             return  t2-t1
...     return wrapper
...
>>>
>>>@timing_fn
...  def my_fun():
...         nlist=[]
...         for num in range(100000):
...             nlist.append(num)
...         print('Sum of all elements : ', sum(nlist))
...
>>> print(my_fn())
```

13. Uses of __and__ in Names

Names that begin and end with two underscores (__) are reserved for use within Python.
Hence, no variable name should have this pattern!

eg: The name of a function is in the system variable 'function.__name__' and its documentation string is 'function.__doc__'

```
>>>def wow():
...         ''' This is a wow!! function
...         Call this again!'''
...         print('This function is named: ', wow.__name__)
...         print('Its docstring is : ', wow.__doc__)
...
>>>

>>>wow()
This function is named: wow
Its docstring is : This is a wow!! function
Call this again!
```

Note: The main program is assigned the special name __main__.
To make a module both import-able and run-able, use the following idiom (at the end of the module):

```
def main():
. . .
. . .

if __name__=='__main__': main()
```

Example:
When your script is run by passing it as a command to the Python interpreter,

```
$python myscript.py
```

Note:
All of the code that is at indentation level 0 gets executed. Functions and classes that are defined are, well, defined, but none of their code gets run. Unlike other languages, there's no main() function that gets run automatically - the main() function is implicitly all the code at the top level.

In this case, the top-level code is an 'if block'. __name__ is a built-in variable which evaluates to the name of the current module. However, if a module is being run directly (as in myscript.py above), then __name__ instead is set to the string "__main__". Thus, you can test whether your script is being run directly or being imported by something else by testing

```
if __name__ == "__main__":
   ...
```

If your script is being imported into another module, its various function and class definitions will be imported and its top-level code will be executed, but the code in the then-body of the if clause above won't get run as the condition is not met. As a basic example, consider the following two scripts:

```
# file one.py
def func():
   print("func() in one.py")

print("top-level in one.py")

if __name__ == "__main__":
   print("one.py is being run directly")
else:
   print("one.py is being imported into another module")


# file two.py
import one

print("top-level in two.py")
one.func()

if __name__ == "__main__":
   print("two.py is being run directly")
else:
   print("two.py is being imported into another module")
```

(1) If you run one.py

$python one.py

Output:

top-level in one.py
one.py is being run directly

(2) If you run two.py

$python two.py

Output:

top-level in one.py
one.py is being imported into another module
top-level in two.py
func() in one.py
two.py is being run directly

Thus, when module one gets loaded, its __name__ equals "one" instead of __main__.

14. Exception basics

Python uses exceptions: code that is executed when an associated error occurs.

It's a good practice to add exception handling anywhere to let the user know what's happening.

In Python, exceptions are processed by 4 statements:

(1)(a) *try/except*
Catch and recover from exceptions raised by Python, or by you
(b) *try/finally*
Perform clean-up actions, whether exceptions occur or not

(2) *raise*
Trigger an exception manually in your code

(3) *assert*
Conditionally trigger an exception in your code

(4) *with/as*
Implement context managers in Python 2.6 onwards

eg: (1) # without exceptions
>>>slist = [1, 2, 3]
>>>pos = 5
>>>slist[pos]
?
error generated by the Python interpreter..not by you!

eg: (2) # with an exception
>>>slist = [1, 2, 3]
>>>pos = 5
>>>try:

```
... slist[pos]
... except:
... print('Position has to be between 0 and ', len(slist)-1, 'but position = ',pos)
...
Position has to be between 0 and 2 but position = 5
```

In Python programs, exceptions are typically used for a variety of purposes. Some of the common roles:

(a) Error handling -
Python raises exceptions whenever it detects errors in programs at runtime. You can catch and respond to the errors in your code, or ignore the exceptions that are raised.
If an error is ignored, Python's default exception-handling behaviour stops the program and prints the error message.
If this default behaviour is not preferred, code a 'try' statement to catch and recover from the exception - Python will jump to 'try' handler when the error is detected, and program resumes execution after try.

(b) Event notification -
Exceptions can also be used to signal valid conditions without you having to pass result flags around a program or test them explicitly.

(c) Special-case handling -
Sometimes a condition may occur so rarely that it's hard to justify convoluting your code to handle it in multiple places. You can eliminate special-case code by handling unusual cases in exception handlers in higher levels of your program. An 'assert' can similarly be used to check that conditions are as expected during development.

(d) Termination actions -
The 'try/finally' statement allows you to guarantee that required closing-time operations will be performed, regardless of the presence or absence of exceptions in your programs. The 'with' statement offers an alternative in for objects that support it.

(e) Unusual control flows -
Exceptions are a sort of high-level and structured 'go to'. They can be used for implementing exotic control flows.
Eg: You can implement 'backtracking' in Python by using exceptions even though it does not explicitly support it.
There is no 'go to' in Python, but exceptions can sometimes serve similar roles, 'raise', for instance, can be used to jump out of multiple loops.

Consider the indexing example as a function..

```
>>>def fetcher(obj,index):
... return obj[index]
...
>>>
>>>x='hello'
>>>fetcher(x,1)
'e'
```

Catching exceptions

```
>>>try:
... fetcher(x, 5)
... except IndexError:
... print('got exception')
...
got exception
```

```
>>>
```

Raising exceptions

```
>>>try:
. . .       raise IndexError
. . . except IndexError:
. . .       print('got exception')
. . .
got exception
```

Termination Actions

```
>>>try:
. . .      fetcher(x, 3)
. . . finally:
. . .      print('after fetch')
. . .
'l'
after fetch
>>>
```

The 'with/as' statement runs an object's context management logic to guarantee that termination actions occur, irrespective of any exceptions i its nested block

```
>>>with open('file1.txt','w') as f1: #always closes the file
. . .       f1.write('The end!\n')
. . .
9
>>>
>>> with open('file1.txt','r') as f1: #always closes the file
. . .         s=f1.read()
. . .
>>>s
'The end!\n'
```

Some examples:

(1) without try..

```
>>> n = int(raw_input("Please enter a number: "))
Please enter a number: 23.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
```

With 'try - except'

```
while True:
   try:
      n = raw_input("Please enter an integer: ")
      n = int(n)
      break
   except ValueError:
      print("No valid integer! Please try again ...")
print "Great, you successfully entered an integer!"
```

Executing..

```
$ python integer_read.py
```

```
Please enter an integer: abc
No valid integer! Please try again ...
Please enter an integer: 42.0
No valid integer! Please try again ...
Please enter an integer: 42
Great, you successfully entered an integer!
$
```

(2) try..finally

```
#finally.py
try:
   x = float(raw_input("Your number: "))
   inverse = 1.0 / x
finally:
   print("There may or may not have been an exception.")
print "The inverse: ", inverse
```

Executing..

```
$ python finally.py
Your number: 34
There may or may not have been an exception.
The inverse:  0.0294117647059

$ python finally.py
Your number: Python
There may or may not have been an exception.
Traceback (most recent call last):
  File "finally.py", line 3, in <module>
   x = float(raw_input("Your number: "))
ValueError: invalid literal for float(): Python
$
```

(3) Combining try, except and finally

```
#finally2.py
try:
   x = float(raw_input("Your number: "))
   inverse = 1.0 / x
except ValueError:
   print "You should have given either an int or a float"
except ZeroDivisionError:
   print "Infinity"
finally:
   print("There may or may not have been an exception.")

$ python finally2.py
Your number: 37
There may or may not have been an exception.

$ python finally2.py
Your number: seven
You should have given either an int or a float
There may or may not have been an exception.

$ python finally2.py
```

Your number: 0
Infinity
There may or may not have been an exception.
$

(4) 'assert' statement

The assert statement is intended for debugging statements. It can be seen as an abbreviated notation for a conditional raise statement, i.e. an exception is only raised, if a certain condition is not True.

Without using the assert statement, we can formulate it like this in Python:
if not <some_test>:
    raise AssertionError(<message>)

The following code, using the assert statement, is semantically equivalent, i.e. has the same meaning:
assert <some_test>, <message>

The line above can be "read" as: If <some_test> evaluates to False, an exception is raised and <message> will be output.

Example:
>>> x = 5
>>> y = 3
>>> assert x < y, "x has to be smaller than y"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: x has to be smaller than y
>>>

Hint: assert should be used for trapping user-defined constraints only and not programming errors (Python does it)!