V. Advance tools

Range()

- We can generate a sequence of numbers using range() function.
- range(10) will generate numbers from 0 to 9 (10 numbers).
- We can also define the start, stop and step size as range(start, stop, step_size). step_size defaults to 1 if not provided.
- This function doesn't store all the values in memory. On-fly it will generate the next number based on start, stop & step-size.
    - range(10)
    - range(-5, 8)
    - range(0, 150, 10)

Xrange()

- This function returns the **generator object** that can be used to display numbers only by looping.
- Only particular range is displayed on demand. So its an "**lazy evaluation**".
- x = xrange(1,100)

```
for i in x:
        print i
```

	range()	xrange()
Return Type	list	Generator object
Memory	Occupies more memory Depends on the elements count	Same memory though it has any no of elements.
Operation Usage	All list operations Slicing possible	Slicing not possible. Indexing works
Speed	Little slow	Faster
Supported versions	Works in both 2.7 & 3.x versions	Only works in 2.7 Deprecated in 3.x

Copy

- Sometimes we may want to have the original values unchanged and only modify the new values or vice versa.
- Python supports two ways to create copies:
    - Shallow Copy
    - Deep Copy
- Use the ***copy*** module

Shallow Copy

- A shallow copy creates a new object which stores the reference of the original elements.
- Shallow copy doesn't create a copy of nested objects/values, instead it just copies the reference of nested objects.
- This copy process does not recurse or create copies of nested objects itself.

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

#Check id of new & old object
#Check id of each nested objects as well
old_list.append([4, 4, 4])

print("Old list:", old_list)
print("New list:", new_list)


old_list[1][1] = 'AA'

print("Old list:", old_list)
print("New list:", new_list)
```

Deep Copy

- A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
- We are going to create deep copy using deepcopy() function present in copy module.
- The deep copy creates independent copy of original object and all its nested objects.
- If we make changes to any nested objects in original object , there will be no changes to the copied object(new one)

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

print("Old list:", old_list)
print("New list:", new_list)

#Check id of new & old object
#Check id of each nested objects as well
old_list[1][0] = 'BB'

print("Old list:", old_list)
print("New list:", new_list)
```

Iterator

Iterator in python is any python type that can be used with a 'for in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement following methods. In fact, any object that wants to be an iterator must implement following methods.

1. **__iter__** method that is called on initialization of an iterator. This should return an object that has a next or __next__ (in Python 3) method.
2. **next (__next__ in Python 3)** The iterator next method should return the next value for the iterable. When an iterator is used with a 'for in' loop, the for loop implicitly calls next() on the iterator object. This method should raise a StopIteration to signal the end of the iteration.

```
my_list = [4, 7, 0, 3]
# get an iterator using iter()
my_iter = iter(my_list)
## iterate through it using next()
#prints 4
print(next(my_iter))
#prints 7
print(next(my_iter))
## next(obj) is same as obj.__next__()
#prints 0
print(my_iter.__next__())
#prints 3
print(my_iter.__next__())
## This will raise error, no items left
next(my_iter)
```

Generator

- Generators offer a comfortable method to create iterators.
- It is like a function. It returns an iterator object.
- Code will not be executed when we call the method/function.
- Yield → returns the value of the expression. Python keeps track of position of the yield statement.
- Next → execution continues from the yield statement

```
def city_generator():
      yield("Karnataka")
      yield("MP")
      yield("Delhi")
      yield("UP")

city_gen_obj = city_generator()
```

```
print type(city_gen_obj)
print dir(city_gen_obj)
print city_gen_obj.next()
print city_gen_obj.next()
print city_gen_obj.next()
print city_gen_obj.next()
```

List, Set & Dict comprehensions

List Comprehensions

- Precise way of creating lists for a list of elements based on some condition.
- Returns a list
- list_comprehenstion = [<expression>   <for loop>   <optional conditional statement>]
- list1 = [x** 2 for x in range(10)]
- list2 = [x**2 for x in range(10) if x%2==0]

Set Comprehensions

- It returns set of elements for the given list of items with satisfied condition(optional)
- It returned the set. Means it unique elements though we pass duplicate elements as an input.
- set_comprehension = {<expression>   <for loop>   <optional condition>}
- set1 = {x*2 for x in range(10)}
- list1= [1,2,3,4,1,5,6,1,3]
- set2 = {x**2 for x in list1 if x >2}

Dict Comprehensions

- Dict comprehensions returns the dict based on the input and the expression.
- Dict_comprehension = {element : <expression>   <for loop>   <optional condition>}
- list1 = [5,15,25,35,45,55]
- dict1 = {x : x**2 for x in list1 }
- dict2 = {x*3 : x**2 for x in range(10,100,5) if x%10==0 }

Map, Filter & Reduce

map()

- Map function maps the given elements with the expression.
- map(function, list) → Function can have some expressions on the element.

filter()

- Filter function used for filtering the given list of elements based on some condition.
- It filters all the elements which satisfy the return of function

reduce()

- Function reduce(function(), list_of_items) continuously applies the function() to all the elements sequentially.
- It returns a single value.

Nested Functions.

There are four cases when we do have function inside the functions.

1. **Functions inside the functions**

```
def f():
    def g():
        print("Hi, it's me 'g'")
        print("Thanks for calling me")
    print("This is the function 'f'")
    print("I am calling 'g' now:")
    g()

f()
```

2. **Functions inside the function with return statement**

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32
    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

print(temperature(20))
```

3. **Function as a parameter to other function**

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

4. **Function as a parameter to other function and returning the function**

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

Decorators

Decorators are used in a situation where user don't want to change the original functionality of a method, he/she wants to modify(add) functionality in wrapper.

Decorator expect function as an argument and returns the wrapper/inside function.

```
def decorator_function(original_function):
    def wrapper_function():
        return original_function()
    return wrapper_function

@decorator_function
def display():
    print "Hi, we are in display function"

display()

@decorator_function
def new_display():
    print "Hello, we are in new_display function"

new_display()
```

Here display() and new_display() both the functions are decorated with decorator_funciton().

Decorators also can have arbitrary and keyword arguments. Below example demonstrate the same.

```
def decorator_function_1(original_function):
    def wrapper_function(*args, **kwargs):
        return original_function(*args, **kwargs)
    return wrapper_function
```

```
@decorator_function_1
def display_student(name, age):
    print("Name of student : {} and age is {}".format(name,age))

display_student("Prem", 16)

@decorator_function_1
def display_arbitrary_args(*args, **kwargs):
    for arg in args:
        print arg
    for key in kwargs:
        print key, kwargs[key]

display_arbitrary_args(4,5,"raja", "Bangalore", 560103)
my_dict = {"name":"raja", "city":"Bangalore", "pin":560103}
display_arbitrary_args(**my_dict)
```

Classes can also act as decorator. Refer below example.

```
#class decorator
class class_decorator(object):
      def __init__(self,original_func):
            self.original_func = original_func

      def __call__(self, *args, **kwargs):
            return self.original_func(*args, **kwargs)

@class_decorator
def display(name, age):
      print("Name of student: {} & his age is {}".format(name, age))

display("Prem", 13)
display("Ravi", 17)
```

File Operations

- A file operation takes place in the following order.
    - Open a file
    - Read or write (perform operation)
    - Close the file
- Python has a built-in function open() to open a file.
- This function returns a file object, also called a handle, as it is used to read or modify the file accordingly
    - f = open("test_file.txt")

File opening modes

- file_handle1 = open("default_read_file.txt")   # equivalent to 'r' or 'rt'
- file_handle2 = open("file_write_mode.txt", 'w') # write in text mode
- file_handle3 = open("file_read_write_binary.bmp", 'r+b') # read and write in binary mode
- When working with files in text mode, it is recommended to specify the encoding type.
    - f = open("test.txt", mode = 'r', encoding = 'utf-8')
    - Default encoding is **'utf-8'**

- Close file
    - file_handle.close()

File opening with *with* keyword

- The best way to do handle files 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

  # statements to perform on files

Write to a file

- In order to write into a file in Python, we need to open it in write **'w'**, append **'a'** or exclusive creation **'x'** mode.
- Need to be careful with the **'w'** mode as it will overwrite into the file if it already exists. **All previous data will be 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.

Reading a file

- we must open the file in reading mode.
- 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')
    - f.read(4) # read the first 4 data
    - f.read(10) # read the next 10 data
    - f.read() # read in the rest till end of file
    - f.read() # further reading returns empty sting

Useful methods in File handling

- f.tell() → get the current file position
- f.seek(0) → bring file cursor to initial position
- f.read() → read entire file
- **read line by line**

  for line in f:

  print line
- f.readline() → Read entire line
- f.readlines() → read entire file line by line. Returns list of lines.

Handling directories

- Python has the ***os*** module, which provides us with many useful methods to work with directories and files.
    - import os
    - os.getcwd() # Get the present working directory
    - os.chdir('C:\\Python27') #change the current working
    - os.listdir() # Returns all files and sub directories inside a directory

- os.mkdir('test') # Create a new directory
- os.rename('old_directory','new_directory') # Rename a directory or a file.
- os.remove('old.txt') # A file can be removed/deleted.
- os.rmdir('new_one') # Removes an empty directory.

Threading

In Python, the **threading** module provides a very simple and intuitive API for spawning multiple threads in a program.

To create a new thread, we create an object of **Thread** class. It takes following arguments:

- **target**: the function to be executed by thread
- **args**: the arguments to be passed to the target function

  t = Thread(target=func, args=(a,b,c))

To start a thread, we use **start** method of **Thread** class.

  t.start()

Once the threads start, the current program also keeps on executing. To stop execution of current program until a thread is complete, we use **join** method.

  t.join()

```
import time
from threading import Thread

def sleeper(i):
    print "thread %d sleeps for 5 seconds" % i
    time.sleep(15)
    print "thread %d woke up" % i

for i in range(100):
    t = Thread(target=sleeper, args=(i,))
    t.start()

def add(a,b):
      print a+b
      time.sleep(5)
      print("successful thread execution")

x = 5
y = 15
for i in range(5):
      t=Thread(target=add, args=(x,y))
      t.start()
      x=x+1
      y+=1

      t.join()
```