Control Flows

In [1]:

```
# variable assignment
a = 10
```

In [2]:

```
a,b = 10,15
a,b
```

Out[2]:

```
(10, 15)
```

In [3]:

```
#swapping numbers
a,b = b, a
a,b
```

Out[3]:

```
(15, 10)
```

In [4]:

```
my_list = []
my_list = 12,18
my_list
```

Out[4]:

```
(12, 18)
```

In [5]:

```
# if conditional statement
my_age = int(raw_input("Enter Your age: "))

if (my_age <= 35):
    print("Young, consider investing in equities")
elif((my_age > 35) and (my_age <= 50)):
    print("Middle age , consider moving to secure instruments")
elif((my_age > 50) and (my_age <= 65)):
    print("You probably might be thiking of retiring now")
else:
     print("Might have retired already!!")
```

```
Enter Your age: 35
Young, consider investing in equities
```

In [6]:

```
# operatos - == < > <= >= n
age = 30
print(age == 30)
print(age <= 30)
print(age < 30)
print(age > 30)
print(age >= 30)
print(age != 30)
```

```
True
True
False
False
True
False
```

In [7]:

```
my_list1 = [2,3,4]
my_list2 = [3,2,4]
my_list3 = [1,2,3]
```

In [8]:

```
my_list1 == my_list2
```

Out[8]:

```
False
```

In [9]:

```
my_list1 == sorted(my_list2)
```

Out[9]:

```
True
```

In [10]:

```
my_list3 < my_list1
```

Out[10]:

```
True
```

In [11]:

```
# for loop
# iterates over the items of any sequence (list, string, tuple,dictinary,set)

number_list = range(1,10)

# find all of the odd numbers between 1 and 10

for number in number_list:
    if (number%2):
        print(number)
```

```
1
3
5
7
9
```

In [12]:

```
# for loop

my_list3 = ['apple','banana','orange','grape','mango']

for i,fruit in enumerate(my_list3):
    if (fruit.upper().startswith('G')):
        print(i,fruit)
        break
else:
    print("There is no fruit that starts with 'g'")
```

```
(3, 'grape')
```

List Comprehensions

In [13]:

```
a = [10,20,30,40,50,60]
a
```

Out[13]:

```
[10, 20, 30, 40, 50, 60]
```

In [14]:

```
new_a = []
incriment = 5
for number in a:
    new_a.append(number + incriment)
new_a
```

Out[14]:

```
[15, 25, 35, 45, 55, 65]
```

In [15]:

```
# using list comprehensions
new_a = [number+5 for number in a]
new_a
```

Out[15]:

```
[15, 25, 35, 45, 55, 65]
```

In [16]:

```
b = [number+5 for number in a if number > 20] # you can filter items also
```

In [17]:

```
b
```

Out[17]:

```
[35, 45, 55, 65]
```

In [18]:

```
name = None
while(name != 'quit'):
    name = str(raw_input("Please enter a string, enter quit"))
    print("you have entered %s\n"%name)
```

```
Please enter a string, enter quitquit
you have entered quit
```

Functions

- The keyword **def** introduces a function definition.
- The **first statement** of the function body can optionally be a string literal;
- this string literal is the function's documentation string, or **docstring**.
- Variable Scope - local, global and then builtin
- Default Argument Values
- Keyword Arguments
- *args and **kwargs

In [19]:

```
# it is always a good practice to provide the docstring
def simple_function():
    """This is a simple functions without any arguments"""
    print("This is a simple function")
```

In [20]:

```
simple_function
```

Out[20]:

```
<function __main__.simple_function>
```

In [21]:

```
simple_function()
```

```
This is a simple function
```

In [22]:

```
simple_function?
```

In [23]:

```
def squares(a):
    """This function returns the square of a given number
       squares(number) -> square of the number"""
    return a*a
```

In [24]:

```
squares(3)
```

Out[24]:

```
9
```

In [25]:

```
squares()
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-25-f52afd3ecd75> in <module>()
----> 1 squares()

TypeError: squares() takes exactly 1 argument (0 given)
```

In [ ]:

```
# Providing a default value
#
def squares(a=2):
    """This function returns the square of a given number
       squares(number=2) -> square of the number"""
    return a*a
```

In [ ]:

```
squares()
```

In [ ]:

```
squares(3)
```

In [ ]:

```
def send_email(to="no_email",subject="No Subject",from_add="Sasi@example.com",bo
dy="no_content"):
    return "To:" + to + ";\nSubject:" + subject +";\nFrom:" + from_add + ";\nbod
y:" + body + "\n"
```

In [ ]:

```
send_email(to="unknown@fmr.com",subject="This is a test message",body="I am in P
ython training")
```

function scope

In [ ]:

```
a = 15
def myfunc(b):
    print 'a:',a
    return a + b
```

In [ ]:

```
myfunc(3)
```

In [ ]:

```
myfunc(5)
```

In [ ]:

```
def newfunc(b):
    a = 5
    print 'a:', a
    return a + b
print 'global a:', a
```

In [ ]:

```
newfunc(3)
```

In [ ]:

```
myfunc(3)
```

In [ ]:

```
def globfunc(b):
    global a
    a = 5
    print 'a:', a
    return a + b
```

In [ ]:

```
globfunc(3)
```

In [ ]:

```
myfunc(5)
```

*args and **kwargs

In [ ]:

```
def test_args(*args,**kwargs):
    for arg in args:
        print(arg)
    for key in kwargs:
        print(key,kwargs[key])
```

In [ ]:

```
test_args(1,3,4,language='Python')
```

In [ ]:

```
input_list = ['a','b','c','d']
input_kwargs = {'language':'Python'}
test_args(*input_list,**input_kwargs)
```

Reading and writing Files

In [ ]:

```
file_handle = open('output.txt','w')
file_handle
```

In [ ]:

```
file_handle.write("This is first line\n")
file_handle.write("This is second line\n")
```

In [ ]:

```
file_handle.close()
```

In [ ]:

```
inp_file = open('output.txt','r')
for line in inp_file:
    print(line.strip("\n"))
inp_file.close()
```

In [ ]:

```
# with takes care of opening and closing the files, there is no explict closing
 is required
with open('output.txt','r') as f:
    print(f.read())
```

In [ ]:

```
data = []
with open('output.txt') as f:
    data = f.readlines()
print(data)
```

Exception handling

In [ ]:

```
a = 100/0.0
```

In [ ]:

```
import sys
```

In [ ]:

```
try:
    a = 100/1.0
    c = b/a
except ZeroDivisionError as e:
    print(e)
except:
    print("Unexpected Error",sys.exc_info()[0])
```

Modules

- #### A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.
- #### Reuse of code
- #### Modularity for maintenance of the code base

In [ ]:

```
# let's check what is there in utilities.py file
# the module name is "utilities"
```

In [ ]:

```
%load utilities.py
```

In [ ]:

```
import utilities
```

In [ ]:

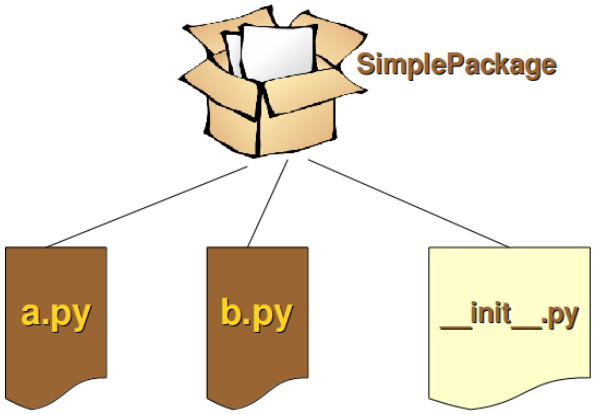
```
dir(utilities)
```

In [ ]:

```
# accessing the functions in utilities module
utilities.hello("yourname")
```

Packages

- #### Packages are a way of structuring Python's module namespace



In [ ]:

```
# Suppose you are creating a software package for restaurant business
# Think, How you want to oganize your code
# What are the various functions necessary
# booking
# billing
# discounts
# menu
```

Classes

- #### Python classes provide all the standard features of Object Oriented Programming
- #### The class inheritance mechanism allows multiple base classes
- #### A derived class can override any methods of its base class or classes

In [ ]:

```
# simple class
class MyClass:
    """This is a sample class"""
    class_variable = 1.2
    def __init__(self,name):
        self.name = name
    def get_name(self):
        return self.name
    def set_name(self,name):
        self.name = name
```

In [ ]:

```
print(MyClass.class_variable)
```

In [ ]:

```
myobj = MyClass('My Object')
```

In [ ]:

```
print(myobj.class_variable)
```

In [ ]:

```
myobj
```

In [ ]:

```
type(myobj)
```

In [ ]:

```
myobj.get_name()
```

In [ ]:

```
myobj.set_name("Object Name Changed")
```

In [ ]:

```
myobj.get_name()
```

In [ ]:

```
# Class inheritance
class MyNewClass(MyClass):
    pass
```

In [ ]:

```
newobj = MyNewClass("New Class Object")
```

In [ ]:

```
# Class inheritance
class MyNewClass(MyClass):
    def __init__(self,name,age):
        MyClass.__init__(self,name)
        self.age = age
```

In [ ]:

```
newobj = MyNewClass("Test",30)
newobj.age
```

Regular Expression

- #### A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

In [ ]:

```
import re
pattern = re.compile('be*')
```

In [ ]:

```
match = pattern.search("This is a beautiful day")
print (match)
```

In [ ]:

```
# some regular expression patterns
# [0-9]+ or \d+ matches a series of digits
# [a-z]+ or \w+ matches a seris of alphabets
#[^a-z0-9A-Z]+ other than alphabet and number
# ^[a-z]{2} starting with 2 alphabets
# [0-9]+$ ends with number
```

In [ ]:

```
pattern1 = re.compile("[\+]*[0-9]{2}[0-9]{10}") # \ is escape character
pattern2 = re.compile("[\+]*[0-9]{10}[78][5]")
```

In [ ]:

```
my_str = "My telehone number is +919980373275, you can contact me at 91938529385
2"
groups1 = pattern1.findall(my_str)
groups2 = pattern2.findall(my_str)
```

In [ ]:

```
print(groups1)
print(groups2)
```

future

- #### With __future__ , you can slowly be accustomed to incompatible changes or to such ones introducing new keywords.
- #### In 2.5, as the with keyword was new and shouldn't be used as variable names any longer
- #### *from __future__ import with_statement* statement is needed In order to be able to use a program which uses variables named with

In [ ]:

```
print(8/7)
print(8//7)
```

In [ ]:

```
from __future__ import division
print(8/7)
print(8//7)
```

Iterator

- #### The built-in function iter takes an iterable object and returns an iterator
- #### Each time we call the next method on the iterator gives us the next element
- #### The __iter__ method is what makes an object iterable
- #### The return value of __iter__ is an iterator. It should have a next method and raise StopIteration when there are no more elements

In [ ]:

```
my_iter = iter([1,2,3]) # returns iterator object and this has next method
my_iter
```

In [ ]:

```
print my_iter.next() # returns 1
print my_iter.next() # returns 2
print my_iter.next() # returns 3
print my_iter.next() # raises StopIteration Exception
```

Generator

- #### Generators simplifies creation of iterators
- #### A generator is a function that produces a sequence of results instead of a single value
- #### yield is a keyword. Each time the yield statement is executed the function generates a new value

In [ ]:

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

In [ ]:

```
a = my_range(5)
print a.next()
print a.next()
```

In [ ]:

```
for i in a:
    print i
```