Classes

```
class MyClass : #at least one line of code should be inside

print (MyClass)
```

```
  File "<ipython-input-29-cbe1166dbf86>", line 3
    print (MyClass)
        ^
IndentationError: expected an indented block
```

SEARCH STACK OVERFLOW

```
class MyClass :
  pass

print (MyClass)
```

```
<class '__main__.MyClass'>
```

```
class MyClass :
  x = 5

print (MyClass) #blue print
```

```
<class '__main__.MyClass'>
```

```
class MyClass :
  x = 5

p1 = MyClass()

print (p1) #actual object

print (p1.x)
```

```
class MyClass:
  x = 5

p1 = MyClass()

p2 = MyClass()

p1.x = 10 #equivalent to having a separate space for all data
p2.x = 20

print (p1.x)
print (p2.x)
```

```
class MyClass :

```

```
  x = 5

  def dosomething() : #incorrect way of definingin (inside a class)
    print ('inside class')


p1 = MyClass()

p1.dosomething() #does not work

```

```
class MyClass :
  x = 5

  def dosomething(self) :
    print ('inside class')

p1 = MyClass()
p1.dosomething(self) #incorrect way of invocation
```

```
class MyClass :
  x = 5

  def dosomething(self) :
    print ('inside class')

p1 = MyClass()
p1.dosomething() #correct way of invocation
```

```
class MyClass :
  x = 5

  def dosomething(self,new_x) :
    x = new_x
    print ('inside class',x)

p1 = MyClass()

```

```
print (p1.x)

p1.dosomething(self,10)

print (p1.x)
```

```
class MyClass :
  x = 5

  def dosomething(self,new_x) :#correct way of stating

    x = new_x #variable is not linked to this method (incorrect results)

    print ('inside class',x)

p1 = MyClass()

print (p1.x)

p1.dosomething(10) #correct way of invocation

print (p1.x)
```

```
class MyClass :
  self.x = 5 #incorrect way of specification

  def dosomething(self,new_x) :#correct way of stating

    self.x = new_x #correct way of specification

    print ('inside class',x)

p1 = MyClass()

print (p1.x)

p1.dosomething(10) #correct way of invocation

print (p1.x)
```

Constructors

First parameter (self)

```
class MyClass :

  def __init__(self) :
    self.x = 5

  def dosomething(self,new_x) :#correct way of stating

    self.x = new_x #correct way of specification

    print ('inside class',self.x)

p1 = MyClass()

print (p1.x)

p1.dosomething(10) #correct way of invocation

print (p1.x)
```

```
class MyClass :

  def __init__(jamesbond) :
    jamesbond.x = 5

  def dosomething(jamesbond,new_x) :#correct way of stating

    jamesbond.x = new_x #correct way of specification

    print ('inside class',jamesbond.x)

p1 = MyClass()

print (p1.x)

p1.dosomething(10) #correct way of invocation

print (p1.x)
```

```
class MyClass :

  def __init__(self,x1,y1) : #with arguments
    self.x = x1
    self.y = y1

  def dosomething(self,new_x, new_y) :#correct way of stating

    self.x = new_x #correct way of specification
    self.y = new_y

    print ('inside class',self.x, self.y)

p1 = MyClass(1,2)

print (p1.x, p1.y)

p1.dosomething(10,20) #correct way of invocation

print (p1.x,p1.y)
```

```
class MyClass :

  def __init__(self,x1,y1) : #with arguments
    self.x = x1
    self.y = y1

  def dosomething(self,new_x, new_y) :#correct way of stating

    self.x = new_x #correct way of specification
    self.y = new_y

    print ('inside class',self.x, self.y)

p1 = MyClass(1,2)

print (p1.x, p1.y)

p1.dosomething(10,20) #correct way of invocation

print (p1.x,p1.y)

p1.x = -100
p1.y = -200

print (p1.x, p1.y) #mddify the values as instance variables of the object
```

Deletion and Creation

```
class MyClass :
  def __init__(self) :
    self.x = 10

p1 = MyClass()

```

```
print (p1.x)

del p1.x #remove an attribute

print (p1.x)
```

```
class MyClass :
  def __init__(self) :
    self.x = 10

p1 = MyClass()

print (p1.x)

del p1 #delete an instance

print (p1.x)
```

```
class MyClass :
  def __init__(self,x1) :
    self.x = x1

p1 = MyClass(10)

print (p1.x)

del p1 #delete an instance

p1 = MyClass(100) #recreate p1

print (p1.x)
```

Inheritence

Simple classes

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1

  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :
  pass

y = Student('James','Bond')

y.disp_name()
```

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1

  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :

  def __init__(self,fname1, lname1) : #incorrect way
    #parent class's method is overridden (think as over-written) and not invoked
    pass

y = Student('James','Bond')

y.disp_name()
```

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1
```

```
  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :

  def __init__(self,fname1, lname1) : #correct way below
    Person.__init__(self,fname1,lname1)

y = Student('James','Bond')

y.disp_name()
```

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1

  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :

  def __init__(self,fname1, lname1) : #correct way below (simplified)
    Person.__init__(self,fname1,lname1)

y = Student('James','Bond')

y.disp_name()
```

super clause

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1

  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :

  def __init__(self,fname1, lname1) : #correct way below (simplified)
    super().__init__(fname1,lname1)

y = Student('James','Bond')

y.disp_name()
```

```
class Person:

  def __init__(self,fname1, lname1) :
    self.fname = fname1
    self.lname = lname1

  def disp_name(self) :
    print (self.fname, self.lname)

class Student(Person) :

  def __init__(self,fname1, lname1, coursename) : #additional parameter
```

```
    self.coursename = coursename
    super().__init__(fname1,lname1)


  def disp_details(self) :
    print (self.fname, self.lname, self.coursename)


y = Student('James','Bond', 'Maths')

y.disp_details()
```

Polymorphism

Not the usual one as in Java language

```
def method1() :
  print ('method1 no params')

def method1(param) :
  print ('method1 single param')

def method1(param1, param2) :
  print ('method1 two params')

method1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-60-8f9a6bca3539> in <module>()
      8   print ('method1 two params')
      9
---> 10 method1()

TypeError: method1() missing 2 required positional arguments: 'param1' and 'param2'
```

SEARCH STACK OVERFLOW

Explicit is better than implict

```
def method1(myobj=None) :
  if myobj['what'] == 'noarg' :
    print ('method with no arguments')
    return
  if myobj['what'] == 'single' :
    print ('method with single argument')
    return

  if myobj['what'] == 'two' :
    print ('method with two arguments')
    return

myparam = {'what':'noarg','other':'data'}

method1(myparam)

myparam = {'what':'single','other':'data'}

```

```
method1(myparam)

myparam = {'what':'two','other':'data'}

method1(myparam)

myparam = {'what':'invalid','other':'data'} #what is invalid setting
method1(myparam) #it does nothing
```

```
method with no arguments
method with single argument
method with two arguments
```

Over-riding

```
class A :
  def method(self) :
    print('inside A')

class B(A) :
  def method(self) :
    print('inside B')


b = B()

b.method() #only B's metod is invoked
```

```
inside B
```

```
class A :
  def method(self,x) :
    print ('inside A ',x)

class B (A) :
  def method(self,x,y) :
    print('inside B ',x,y)

b = B()

b.method(1) #wrong invocation
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-3ef3a9ddaa39> in <module>()
      9 b = B()
     10
---> 11 b.method(1)

TypeError: method() missing 1 required positional argument: 'y'
```

SEARCH STACK OVERFLOW

```
class A :
  def method(self,x) :
    print ('inside A ',x)

class B (A) :
  def method(self,x,y) :
    print('inside B ',x,y)

b = B()

```

```
b.method(1,2) #correct way
```

```
inside B  1 2
```

Generators and Iterators

Simple iterator

```
mytuple = "apple", "banana", "cherry"
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
mytuple = ["apple", "banana", "cherry"]
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
mytuple = {'key1':'val1','key2':'val2','key3':'val3'}
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

Customized iterator

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

Generator

```
# A simple generator function
def my_gen():
    a = 10
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n, a, a**n

    n += 1
    print('This is printed second')
    yield n, a-1, (a-1)**n

    n += 1
    print('This is printed at last')
    yield n, a-2, (a-2)**n

# Using for loop
for item in my_gen():
    print(item)
```

Customized Exceptions

REF -https://docs.python.org/3/tutorial/errors.html

Pre-defined exceptions

```
try :
  1/0
except ZeroDivisionError :
  print ('problematic code')
```

```
try :
  1/0
except NameError :
  print ('name exception')
except ZeroDivisionError :
  print ('division exception')
```

```
try :
  1/0
except NameError :
  print ('name exception')
except  : #catches all exceptions
  print ('division exception')

finally :
  print ('closing operations')
```

```
try :
  4 + 3 * spam
except NameError :
  print ('name exception')
except  : #catches all exceptions
  print ('division exception')

finally :
  print ('closing operations')
```

```
try :
  4 + 3 * spam
except (NameError, ZeroDivisionError) :
  print ('name or division exception')
except  : #catches all exceptions
  print ('any exception')

finally :
  print ('closing operations')
```

Custom Exceptions

```
class MyException(Exception) :
  pass

try :
  raise MyException
except Exception as e :
  print (e)
```

```
class MyException(Exception) :

  def __str__(self) :
    return 'custom exception'

try :
  raise MyException
except Exception as e :
  print (e)
```

```
10
```

```
class MyException(Exception) :

  def __init__(self,x) :
    self.x = x

  def __str__(self) :
    return 'custom exception x=' + str(x)

try :
  raise MyException(10)
except Exception as e :
  print (e)
```

Elementary Data Structures

Stack

```
class MyStack :

  def __init__(self,name) :
    self.name = name
    self.mylist = []

  def push(self,x) :
    self.mylist = self.mylist + [x]

  def pop(self) :
    if len(self.mylist) > 0 :
      x = self.mylist[-1]
      del self.mylist[-1]
      return x

  def display(self) :
    print (self.name, self.mylist)

a = MyStack('StackA')
b = MyStack('StackB')

a.push(1)
a.push(2)
a.push(3)

a.display()

x = a.pop()
print (x)
a.display()

b.push(100)
b.push(200)

b.pop()
b.pop()
b.pop()
b.pop()

a.display()
b.display()
```

```
42
```

Queue

```
class MyQueue :

  def __init__(self,name) :
    self.name = name
    self.mylist = []

  def enqueue(self,x) :
    self.mylist = self.mylist + [x]

  def dequeue(self) :
    if len(self.mylist) > 0 :
      x = self.mylist[0]
      del self.mylist[0]
      return x

  def display(self) :
    print (self.name, self.mylist)

a = MyQueue('QueueA')
b = MyQueue('QueueB')

a.enqueue(1)
a.enqueue(2)
a.enqueue(3)

a.display()

x = a.dequeue()
print (x)
a.display()

b.enqueue(100)
b.enqueue(200)

b.dequeue()
b.dequeue()
b.dequeue()
b.dequeue()

a.display()
b.display()

```

Tree

```
class MyTree :

  def __init__(self,x) :
```

```
    self.root = x
    self.left = None
    self.right = None
    print ('created ',x)

  def insert(self,x) :

    if x < self.root :
      if self.left is None :
        self.left = MyTree(x)
      else :
        self.left.insert(x)

    if x >= self.root :
      if self.right is None :
        self.right = MyTree(x)
      else :
        self.right.insert(x)



  def display(self,mystr) :

    print (mystr,self.root)

    if self.left is not None :
      self.left.display(mystr+',<-')

    if self.right is not None :
      self.right.display(mystr+',->')



root = MyTree(4)

for x in [2,1,3,6,5,7] :
  root.insert(x)


root.display('.')
```