→ Classes

```
1 class MyClass : #at least one line of code should be inside
 3 print (MyClass)
      File "<ipython-input-29-cbe1166dbf86>", line 3
Г⇒
        print (MyClass)
    IndentationError: expected an indented block
     SEARCH STACK OVERFLOW
 1 class MyClass:
     pass
 4 print (MyClass)
C→ <class '__main__.MyClass'>
 1 class MyClass:
 4 print (MyClass) #blue print
┌⇒ <class '__main__.MyClass'>
 1 class MyClass :
    x = 5
 4 p1 = MyClass()
 6 print (p1) #actual object
 8 print (p1.x)
C→
 1 class MyClass:
    x = 5
 4 p1 = MyClass()
 6 p2 = MyClass()
 8 p1.x = 10 #equivalent to having a separate space for all data
 9 p2.x = 20
10
11 print (p1.x)
12 print (p2.x)
\Box
 1 class MyClass:
```

```
3
     x = 5
 4
 5
     def dosomething() : #incorrect way of definingin (inside a class)
 6
        print ('inside class')
 7
 8
 9 p1 = MyClass()
10
11 p1.dosomething() #does not work
12
\Box
 1 class MyClass:
 3
 4
     def dosomething(self) :
       print ('inside class')
 5
 7 p1 = MyClass()
 8 p1.dosomething(self) #incorrect way of invocation
C→
 1 class MyClass:
 2
     x = 5
     def dosomething(self) :
  print ('inside class')
 4
 5
 7 p1 = MyClass()
 8 p1.dosomething() #correct way of invocation
\Box
 1 class MyClass:
     x = 5
 3
```

4

5

6

8 p1 = MyClass()
9

def dosomething(self,new_x) :

x = new_x
print ('inside class',x)

```
10 | print (p1.x)
11
12 | p1.dosomething(self,10)
13
14 | 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)
```

1 class MyClass: self.x = 5 #incorrect way of specification 3 def dosomething(self,new_x) :#correct way of stating 4 5 6 self.x = new_x #correct way of specification 7 print ('inside class',x) 9 10 p1 = MyClass() 11 12 print (p1.x) 13 14 p1.dosomething(10) #correct way of invocation 15 16 print (p1.x)

 \Box

Constructors

▼ First parameter (self)

1 class MyClass :

4

5 6

 \Box

def __init__(self) :

self.x = 5

```
8
       self.x = new_x #correct way of specification
       print ('inside class',self.x)
10
11
12 p1 = MyClass()
13
14 print (p1.x)
15
16 p1.dosomething(10) #correct way of invocation
17
18 print (p1.x)
С>
 1 class MyClass :
 3
     def __init__(jamesbond) :
   jamesbond.x = 5
 4
 5
 6
     def dosomething(jamesbond,new_x) :#correct way of stating
 7
 8
       jamesbond.x = new_x #correct way of specification
 9
       print ('inside class',jamesbond.x)
10
11
12 p1 = MyClass()
13
14 print (p1.x)
15
16 p1.dosomething(10) #correct way of invocation
17
18 print (p1.x)
```

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

```
1 class MyClass:
 3
     def __init__(self,x1,y1) : #with arguments
 4
       self.x = x1
 5
       self.y = y1
 6
 7
     def dosomething(self,new_x, new_y) :#correct way of stating
 8
 9
       self.x = new_x #correct way of specification
       self.y = new_y
10
11
       print ('inside class',self.x, self.y)
12
13
14 p1 = MyClass(1,2)
15
16 print (p1.x, p1.y)
17
18 pl.dosomething(10,20) #correct way of invocation
19
20 print (p1.x,p1.y)
L→
 1 class MyClass:
 3
     def init (self,x1,y1) : #with arguments
 4
       self.x = x1
 5
       self.y = y1
 6
 7
     def dosomething(self,new_x, new_y) :#correct way of stating
 8
 9
       self.x = new_x #correct way of specification
10
       self.y = new_y
11
       print ('inside class',self.x, self.y)
12
13
14 p1 = MyClass(1,2)
15
16 print (p1.x, p1.y)
17
18 pl.dosomething(10,20) #correct way of invocation
19
20 print (p1.x,p1.y)
21
22 | p1.x = -100
23 p1.y = -200
24
25 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()
```

```
7 print (p1.x)
8
9 del p1.x #remove an attribute
10
11 print (p1.x)

□→
```

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

p1 = MyClass()

print (p1.x)

del p1 #delete an instance
print (p1.x)
```

```
1  class MyClass :
2   def __init__(self,x1) :
3    self.x = x1
4    p1 = MyClass(10)
6    print (p1.x)
8    del p1 #delete an instance
10    p1 = MyClass(100) #recreate p1
12    print (p1.x)
```

▼ Inheritence

С→

▼ Simple classes

```
1 class Person:
 3
      def __init__(self,fname1, lname1) :
        self.fname = fname1
 4
        self.lname = lname1
 5
 6
 7
      def disp_name(self) :
        print (self.fname, self.lname)
 9
10 class Student(Person) :
11
12
13 y = Student('James', 'Bond')
15 y.disp_name()
C→
 1 class Person:
           _init__(self,fname1, lname1):
 3
        self.fname = fname1
        self.lname = lname1
 5
 6
 7
     def disp_name(self) :
        print (self.fname, self.lname)
 9
10 class Student(Person) :
11
      def __init__(self,fname1, lname1) : #incorrect way
    #parent class's method is overridden (think as over-written) and not invoked
12
13
14
        pass
16 y = Student('James', 'Bond')
17
18 y.disp_name()
C→
```

```
1 class Person:
2
3  def __init__(self,fname1, lname1):
4   self.fname = fname1
5  self.lname = lname1
```

```
6
   7
       def disp_name(self) :
         print (self.fname, self.lname)
  8
  9
  10 class Student(Person) :
  11
  12
       def __init__(self,fname1, lname1) : #correct way below
  13
         Person.__init__(self,fname1,lname1)
  14
  15 y = Student('James', 'Bond')
  16
  17 y.disp_name()
 \Box
   1 class Person:
             _init__(self,fname1, lname1) :
   3
       def
         self.fname = fname1
   4
   5
         self.lname = lname1
   6
   7
       def disp name(self) :
   8
         print (self.fname, self.lname)
   9
  10 class Student(Person) :
  11
            _init__(self,fname1, lname1) : #correct way below (simplified)
  12
         Person.__init__(self,fname1,lname1)
  13
  14
  15 y = Student('James', 'Bond')
  16
  17 y.disp_name()
super clause
   1 class Person:
   3
             _init__(self,fname1, lname1) :
         self.fname = fname1
   5
         self.lname = lname1
   6
   7
       def disp_name(self) :
   8
         print (self.fname, self.lname)
  10 class Student(Person) :
  11
  12
       def __init__(self,fname1, lname1) : #correct way below (simplified)
         super().__init__(fname1,lname1)
  13
  14
  15 y = Student('James', 'Bond')
  16
  17 y.disp_name()
 \Box
   1 class Person:
       def __init__(self,fname1, lname1) :
   3
         self.fname = fname1
   5
         self.lname = lname1
   6
   7
       def disp_name(self) :
   8
         print (self.fname, self.lname)
   9
  10 class Student(Person) :
  11
       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

```
1 def method1() :
     print ('method1 no params')
 3
 4 def method1(param) :
     print ('method1 single param')
 7 def method1(param1, param2) :
     print ('method1 two params')
10 method1()
Гэ
    TypeError
                                                 Traceback (most recent call last)
    <ipython-input-60-8f9a6bca3539> in <module>()
              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

```
1 def method1(myobj=None) :
     if myobj['what'] == 'noarg' :
       print ('method with no arguments')
 3
 4
       return
     if myobj['what'] == 'single' :
 5
       print ('method with single argument')
 6
 7
       return
 8
     if myobj['what'] == 'two' :
10
       print ('method with two arguments')
11
12
13 myparam = {'what':'noarg','other':'data'}
14
15 method1(myparam)
16
17 myparam = {'what':'single','other':'data'}
18
```

```
19 method1(myparam)
    20
    21 myparam = {'what':'two','other':'data'}
    22
    23 method1(myparam)
    24
    25 myparam = {'what':'invalid','other':'data'} #what is invalid setting
    26 method1(myparam) #it does nothing
       method with no arguments
        method with single argument
        method with two arguments
Over-riding
     1 class A:
         def method(self) :
           print('inside A')
     4
     5 class B(A):
     6
         def method(self) :
           print('inside B')
     8
    10 b = B()
    12 b.method() #only B's metod is invoked
       inside B
     1 class A:
         def method(self,x) :
           print ('inside A ',x)
     3
     4
     5 class B (A):
         def method(self,x,y) :
           print('inside B ',x,y)
     8
     9 | b = B()
    10
    11 b.method(1) #wrong invocation
    \Box
                                                      Traceback (most recent call last)
        TypeError
        <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
     1 class A:
         def method(self,x) :
  print ('inside A ',x)
     4
     5 class B (A):
         def method(self,x,y):
     6
           print('inside B ',x,y)
     8
     9 b = B()
    10
```

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

☐→ inside B 1 2
```

Generators and Iterators

▼ Simple iterator

```
1 mytuple = "apple", "banana", "cherry"
 2 myit = iter(mytuple)
 4 print(next(myit))
 5 print(next(myit))
 6 print(next(myit))
□→ apple
    banana
    cherry
 1 mytuple = ("apple", "banana", "cherry")
 2 myit = iter(mytuple)
 4 print(next(myit))
 5 print(next(myit))
 6 print(next(myit))
\Box
 1 mytuple = ["apple", "banana", "cherry"]
 2 myit = iter(mytuple)
 4 print(next(myit))
 5 print(next(myit))
 6 print(next(myit))
C→
 1 mytuple = {'key1':'val1','key2':'val2','key3':'val3'}
 2 myit = iter(mytuple)
 4 print(next(myit))
 5 print(next(myit))
 6 print(next(myit))
С→
 1 mytuple = ("apple", "banana", "cherry")
 3 for x in mytuple:
     print(x)
```

Customized iterator

```
class MyNumbers:
def __iter__(self):
self.a = 1
        return self
 5
 6
      def __next__(self):
    x = self.a
 7
         self.a += 1
 8
 9
         return x
10
11 myclass = MyNumbers()
12 myiter = iter(myclass)
14 print(next(myiter))
15 print(next(myiter))
16 print(next(myiter))
17 print(next(myiter))
18 print(next(myiter))
```

С→

```
1 class MyNumbers:
     def __iter__(self):
    self.a = 1
 4
        return self
 5
     def __next__(self):
   if self.a <= 20:</pre>
 6
 7
 8
          x = self.a
 9
          self.a += 1
10
          return x
11
        else:
12
          raise StopIteration
13
14 myclass = MyNumbers()
15 myiter = iter(myclass)
16
17 for x in myiter:
18
      print(x)
```

Generator

```
1 # A simple generator function
 2 def my_gen():
 3
       a = 10
       n = 1
       print('This is printed first')
 5
       # Generator function contains yield statements
 6
 7
       yield n, a, a**n
 9
       n += 1
10
       print('This is printed second')
       yield n, a-1, (a-1)**n
11
12
13
       print('This is printed at last')
14
15
       yield n, a-2, (a-2)**n
17 # Using for loop
18 for item in my_gen():
19
       print(item)
С→
```

Customized Exceptions

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

▼ Pre-defined exceptions

```
1 try :
2   1/0
3   except ZeroDivisionError :
4   print ('problematic code')

1 try :
2   1/0
3   except NameError :
4   print ('name exception')
5   except ZeroDivisionError :
6   print ('division exception')
```

```
С→
 1 try:
    1/0
 3 except NameError :
    print ('name exception')
 5 except : #catches all exceptions
    print ('division exception')
 8 finally:
     print ('closing operations')
\Box
 1 try:
 2 4 + 3 * spam
 3 except NameError :
    print ('name exception')
 5 except : #catches all exceptions
    print ('division exception')
 8 finally :
9  print ('closing operations')
\Box
 1 try:
    4 + 3 * spam
 3 except (NameError, ZeroDivisionError) :
 4 print ('name or division exception')
5 except : #catches all exceptions
     print ('any exception')
 8 finally:
     print ('closing operations')
\Box
```

Custom Exceptions

```
1 class MyException(Exception) :
     pass
 4 try:
     raise MyException
 6 except Exception as e :
     print (e)
\Box
 1 class MyException(Exception) :
      def __str__(self) :
   return 'custom exception'
 3
 4
 6 try:
 7
     raise MyException
 8 except Exception as e :
     print (e)
```

```
10
C→
 1 class MyException(Exception) :
 3
     def __init__(self,x) :
 4
       self.x = x
 5
 6
     def __str__(self) :
       return 'custom' exception x=' + str(x)
 7
 8
 9
   try:
     raise MyException(10)
10
11 except Exception as e :
     print (e)
C→
```

Elementary Data Structures

→ Stack

```
1 class MyStack :
 3
     def __init__(self,name) :
       self.name = name
 4
 5
       self.mylist = []
 6
 7
     def push(self,x) :
 8
       self.mylist = self.mylist + [x]
     def pop(self) :
10
       if len(self.mylist) > 0 :
11
12
         x = self.mylist[-1]
13
         del self.mylist[-1]
14
         return x
15
     def display(self) :
16
       print (self.name, self.mylist)
17
19 a = MyStack('StackA')
20 b = MyStack('StackB')
21
22 a.push(1)
23 a.push(2)
24 a.push(3)
25
26 a.display()
27
28 x = a.pop()
29 print (x)
30 a.display()
31
32 b.push(100)
33 b.push(200)
34
35 b.pop()
36 b.pop()
37 b.pop()
38 b.pop()
39
40 a.display()
41 b.display()
```

С→

▼ Queue

```
1 class MyQueue :
 3
      def __init__(self,name) :
    self.name = name
 4
 5
        self.mylist = []
 6
 7
      def enqueue(self,x) :
 8
        self.mylist = self.mylist + [x]
 9
10
      def dequeue(self) :
        if len(self.mylist) > 0 :
   x = self.mylist[0]
11
12
           del self.mylist[0]
13
14
           return x
15
      def display(self) :
  print (self.name, self.mylist)
16
17
18
19 a = MyQueue('QueueA')
20 b = MyQueue('QueueB')
22 a.enqueue(1)
23 a.enqueue(2)
24 a.enqueue(3)
25
26 a.display()
27
28 \times = a.dequeue()
29 print (x)
30 a.display()
31
32 b.enqueue(100)
33 b.enqueue(200)
34
35 b.dequeue()
36 b.dequeue()
37 b.dequeue()
38 b.dequeue()
39
40 a.display()
41 b.display()
42
```

▼ Tree

С→

```
1 class MyTree :
2
3 def __init__(self,x) :
```

```
4
       self.root = x
 5
       self.left = None
 6
       self.right = None
 7
       print ('created ',x)
 8
 9
     def insert(self,x) :
10
11
       if x < self.root :</pre>
12
         if self.left is None :
13
           self.left = MyTree(x)
14
         else:
           self.left.insert(x)
15
16
       if x \ge self.root:
17
         if self.right is None:
18
19
           self.right = MyTree(x)
20
         else :
           self.right.insert(x)
21
22
23
24
25
     def display(self,mystr) :
26
27
       print (mystr,self.root)
28
29
       if self.left is not None :
         self.left.display(mystr+',<-')</pre>
30
31
       if self.right is not None :
32
33
         self.right.display(mystr+',->')
34
35
36
37 root = MyTree(4)
38
39 for x in [2,1,3,6,5,7]:
40
     root.insert(x)
41
42
43 root.display('.')
44
45
```

