



# PROGRAMMING IN PYTHON

Gavrilit Dragos  
Course 10

# CLASSES (INHERITANCE)

Python classes supports both simple and multiple inheritance.

Python 3.x (simple inheritance)

```
class <name> (Base) :  
    <statement1>  
    ...  
    <statementn>
```

Where **statement<sub>i</sub>** is usually a declaration of a method or data member.

Python 3.x (multiple inheritance)

```
class <name> (Base1, Base2, ... Basem) :  
    <statement1>  
    ...  
    <statementn>
```

# CLASSES (INHERITANCE)

Python has two keywords (**issubclass** and **isinstance**) that can be used to check if an object is a subclass of an instance of a specific type.

Python 3.x (simple inheritance)

```
class Base:
    x = 10
class Derived(Base):
    y = 20

d = Derived()
print ("d.X = ",d.x)
print ("d.Y = ",d.y)
print ("Instance of Derived:",isinstance(d,Derived))
print ("Instance of Base:",isinstance(d,Base))
print ("Derived is a subclass of Base:",issubclass(Derived,Base))
print ("Base is a subclass of Derived:",issubclass(Base,Derived))
```

## Output

```
d.X = 10
d.Y = 20
Instance of Derived: True
Instance of Base: True
Derived is a subclass of Base: True
Base is a subclass of Derived: False
```

# CLASSES (INHERITANCE)

Inheritance does not assume that the `__init__` function is automatically called for the base when the derived object is created.

Python 3.x (simple inheritance)

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

```
class Derived(Base):
    def __init__(self):
        self.y = 20
```

```
d = Derived()
print ("d.X = ", d.x)
print ("d.Y = ", d.y)
```

Execution error – d.X does not exist because base.\_\_init\_\_ was never called

# CLASSES (INHERITANCE)

Inheritance does not assume that the `__init__` function is automatically called for the base when the derived object is created.

Python 3.x (simple inheritance)

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

class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.y = 20
```

In Python 3 you can also write  
`super().__init__()`

```
d = Derived()
print ("d.X = ", d.x)
print ("d.Y = ", d.y)
```

## Output

```
d.X = 10
d.Y = 20
```

# CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

Python 3.x (simple inheritance)

```
class Base:
    def Print(self):
        print("Base class")

class Derived(Base):
    def Print(self):
        print("Derived class")

d = Derived()
d.Print()
```

**Output**

Derived class

# CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

Python 3.x (simple inheritance)

```
class Base:  
    def Print(self,value):  
        print("Base class",value)
```

```
class Derived(Base):  
    def Print(self):  
        print("Deri
```

```
d = Derived()  
d.Print()  
d.Print(100)
```

**Print** function from Base class was completely overwritten by **Print** function from the derived class. The code will produce a runtime error.

# CLASSES (INHERITANCE)

Inheriting from a class will overwrite all base class members (methods or data members).

In this case member “x” from Base class will be overwritten by member “x” from the derived class.

Python 3.x (simple inheritance)

```
class Base:
    x = 10
class Derived(Base):
    x = 20

d = Derived()
print (d.x)
```

**Output**

20



# CLASSES (INHERITANCE)

Polymorphism works in a similar way. In reality the inheritance is not necessary to accomplish polymorphism in Python.

Python 3.x (simple inheritance)

```
class Forma:
    def PrintName(self): pass
class Square(Forma):
    def PrintName(self): print("Square")
class Circle(Forma):
    def PrintName(self): print("Circle")
class Rectangle(Forma):
    def PrintName(self): print("Rectangle")

for form in [Square(), Circle(), Rectangle()]:
    form.PrintName()
```

## Output

```
Square
Circle
Rectangle
```

# CLASSES (INHERITANCE)

Polymorphism works in a similar way. In reality the inheritance is not necessary to accomplish polymorphism in Python.

Python 3.x (simple inheritance)

```
class Square:
    def PrintName(self): print("Square")
class Circle:
    def PrintName(self): print("Circle")
class Rectangle:
    def PrintName(self): print("Rectangle")

for form in [Square(),Circle(),Rectangle()]:
    form.PrintName()
```

## Output

```
Square
Circle
Rectangle
```

# CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
    def MyFunction(self):
        print ("Base A")
class BaseB:
    def MyFunction(self):
        print ("Base B")
class Derived(BaseA, BaseB):
    pass

d = Derived()
d.MyFunction()
```

**Output**

Base A

# CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
    def MyFunction(self):
        print ("Base A")
```

```
class BaseB:
    def MyFunction(self):
        print ("Base B")
```

```
class Derived(BaseA, BaseB):
    pass
```

```
d = Derived()
d.MyFunction()
```



First **MyFunction** from **BaseB**  
is added to **Derived** class

# CLASSES (INHERITANCE)

In case of multiple inheritance, Python derives from the right most class to the left most class from the inheritance list.

Python 3.x (multiple inheritance)

```
class BaseA:
    def MyFunction(self):
        print ("Base A")

class BaseB:
    def MyFunction(self):
        print ("Base B")

class Derived(BaseA, BaseB):
    pass
```

```
d = Derived()
d.MyFunction()
```

Then **MyFunction** from class **BaseA** will overwrite **MyFunction** from **BaseB**

# CLASSES (SPECIAL METHODS)

If we reverse the order (BaseB will be first and BaseA will be the last one), MyFunction will print “Base B” instead of “Base A”

Python 3.x (multiple inheritance)

```
class BaseA:
    def MyFunction(self):
        print ("Base A")

class BaseB:
    def MyFunction(self):
        print ("Base B")

class Derived(BaseB, BaseA):
    pass

d = Derived()
d.MyFunction()
```

**Output**

Base B

# CLASSES (SPECIAL METHODS)

Python defines a special set of functions that can be use do add additional properties to a class. Just like the initialization function (`__init__`) , these functions start and end with “`__`”.

Function	Purpose
<code>__repr__</code> , <code>__str__</code>	Called when the object needs to be converted into string
<code>__lt__</code> , <code>__le__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code>	Operators used to compare instances of the same class.
<code>__bool__</code>	To evaluate the truth value of an object (instance of a class)
<code>__getattr__</code> , <code>__getattribute__</code>	For attribute look-ups
<code>__setattr__</code> , <code>__delattr__</code> <code>__set__</code> , <code>__get__</code>	For attribute operations
<code>__len__</code> , <code>__del__</code> ,	For len / del operators
<code>__setitem__</code> , <code>__getitem__</code> , <code>__contains__</code> , <code>__reversed__</code> , <code>__iter__</code> , <code>__next__</code>	Iterator operators

# CLASSES (SPECIAL METHODS)

Python also defines a set of mathematical functions that can be used for the same purpose:

- ❖ `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`,  
`__pow__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`
- ❖ `__radd__`, `__rsub__`, `__rmul__`, `__rmatmul__`, `__rtruediv__`, `__rfloordiv__`, `__rmod__`, `__rdivmod__`,  
`__rpow__`, `__rlshift__`, `__rrshift__`, `__rand__`, `__rxor__`, `__ror__`,
- ❖ `__iadd__`, `__isub__`, `__imul__`, `__imatmul__`, `__itruediv__`, `__ifloordiv__`, `__imod__`, `__ipow__`,  
`__ilshift__`, `__irshift__`, `__iand__`, `__ixor__`, `__ior__`
- ❖ `__neg__`, `__int__`, `__float__`, `__round__`



# CLASSES (SPECIAL METHODS)

Converting a class to a string. It is recommended to overwrite both `__str__` and `__repr__`

Python 3.x

```
class Test:  
    x = 10
```

```
class Test2:  
    x = 10
```

```
    def __str__(self): return "Test2 with X = "+str(self.x)
```

```
t = Test()  
t2 = Test2()  
print (t, ":", str(t))  
print (t2, ":", str(t2))
```

## Output (Python 3)

```
<__main__.Test object at 0x..> : <__main__.Test object at 0x..>  
Test2 with X = 10 : Test2 with X = 10
```

# CLASSES (SPECIAL METHODS)

Converting to an integer value.

Python 3.x

```
class Test:
    x = 10

class Test2:
    x = 10
    def __int__(self): return self.x
```

```
t = Test()
t2 = Test2()
Value = int(t)
```

This code will produce a runtime error because Python don't know how to translate an object of type Test to an integer

# CLASSES (SPECIAL METHODS)

Converting to an integer value.

Python 3.x

```
class Test:
    x = 10

class Test2:
    x = 10
    def __int__(self): return self.x

t = Test()
t2 = Test2()
Value = int(t2)
```

This code works, **Value** will be 10

# CLASSES (SPECIAL METHODS)

Iterating through a class instance

Python 3.x

```
class CarList:
    cars = ["Dacia", "BMW", "Toyota"]
    def __iter__(self):
        self.pos = -1
        return self
    def __next__(self):
        self.pos += 1
        if self.pos==len(self.cars): raise StopIteration
        return self.cars[self.pos]

c = CarList()
for i in c:
    print (i)
```

**Output (Python 3)**

Dacia  
BMW  
Toyota

# CLASSES (SPECIAL METHODS)

Using class operators. In this case we overwrite `__eq__` (`==`) operator.

Python 3.x

```
class Number:
    def __init__(self, value):
        self.x = value
    def __eq__(self, obj):
        return self.x+obj.x == 0

n1 = Number(-5)
n2 = Number(5)
n3 = Number(6)
print (n1==n2)
print (n1==n3)
```

**Output**

True  
False

# CLASSES (SPECIAL METHODS)

Overwriting the “in” operator (`__contains__`).

Python 3.x

```
class Number:
    def __init__(self, value):
        self.x = value
    def __contains__(self, value):
        return str(value) in str(self.x)

n = Number(123)
print (12 in n)
print (5 in n)
print (3 in n)
```

## Output

```
True
False
True
```

# CLASSES (SPECIAL METHODS)

Overwriting the “len” operator (`__len__`).

Python 3.x

```
class Number:
    def __init__(self, value):
        self.x = value
    def __len__(self, value):
        return len(str(self.x))
```

```
n1 = Number(123)
n2 = Number(99999)
n3 = Number(2)
print (len(n1), len(n2), len(n3))
```

Output

3 5 1

# CLASSES (SPECIAL METHODS)

Building your own dictionary (overwrite `__setitem__` and `__getitem__`)

Python 3.x

```
class MyDict:
    def __init__(self): self.data = []
    def __setitem__(self, key, value): self.data += [(key, str(value))]
    def __getitem__(self, key):
        for i in self.data:
            if i[0]==key:
                return i[1]

d = MyDict()
d["test"] = "python"
d["numar"] = 123
print (d["test"],d["numar"])
```

**Output**

python 123



# CLASSES (SPECIAL METHODS)

Building a bit set (overloading operator [])

Python 3.x

```
class BitSet:
    def __init__(self): self.value = 0
    def __setitem__(self, index, value):
        if value: self.value |= (1 << (index & 31))
        else: self.value -= (self.value & (1 << (index & 31)))
    def __getitem__(self, key):
        return (self.value & (1 << (index & 31))) != 0

b = BitSet()
b[0] = True
b[2] = True
b[4] = True
for i in range(0, 8):
    print("Bit ", i, " is ", b[i])
```

## Output

Bit	0	is	True
Bit	1	is	False
Bit	2	is	True
Bit	3	is	False
Bit	4	is	True
Bit	5	is	False
Bit	6	is	False
Bit	7	is	False

# CONTEXT MANAGER

A context manager is a mechanism where an object is created and notification about the moment that object is being accessed and the moment that object is being terminated.

Context managers are used along with **with** keyword. The objects that are available in a context manager should implement `__enter__` and `__exit__` methods.

```
with item1 as alias1, [item2 as alias2 , ... itemn as aliasn ]:  
    <statement 1 >  
    <statement 2>  
    ....  
    <statement n>
```

```
with item1, [item2, ... itemn]:  
    <statement 1 >  
    <statement 2>  
    ....  
    <statement n>
```

# CONTEXT MANAGER

Whenever a **with** command is encounter, the following steps happen:

1. All items are evaluated
2. For all items `__enter__` is called
3. If aliases are provided, the result of the `__enter__` method is store into the alias
4. The block within the **with** is executed
5. If an exception appears, `__exit__` is called and information related to the exception (type, value and traceback) are provided as parameters. If the `__exit__` method returns false, the exception is re-raised. If the `__exit__` method returns true, the exception is ignored.
6. If no exception appear, `__exit__` is called with None parameters for (type, value and traceback). The result from the `__exit__` method will be ignored.

# CLASSES (SPECIAL METHODS)

## File context manager

Python 3.x

```
class CachedFile:
    def __init__(self, fileName):
        self.data = ""
        self.fileName = fileName

    def __enter__(self):
        print("__enter__ is called")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("__exit__ is called")
        open(self.fileName, "wt").write(self.data)
        return False

with CachedFile("Test.txt") as f:
    f.data = "Python course"
```

### Output

```
__enter__ is called
__exit__ is called
```