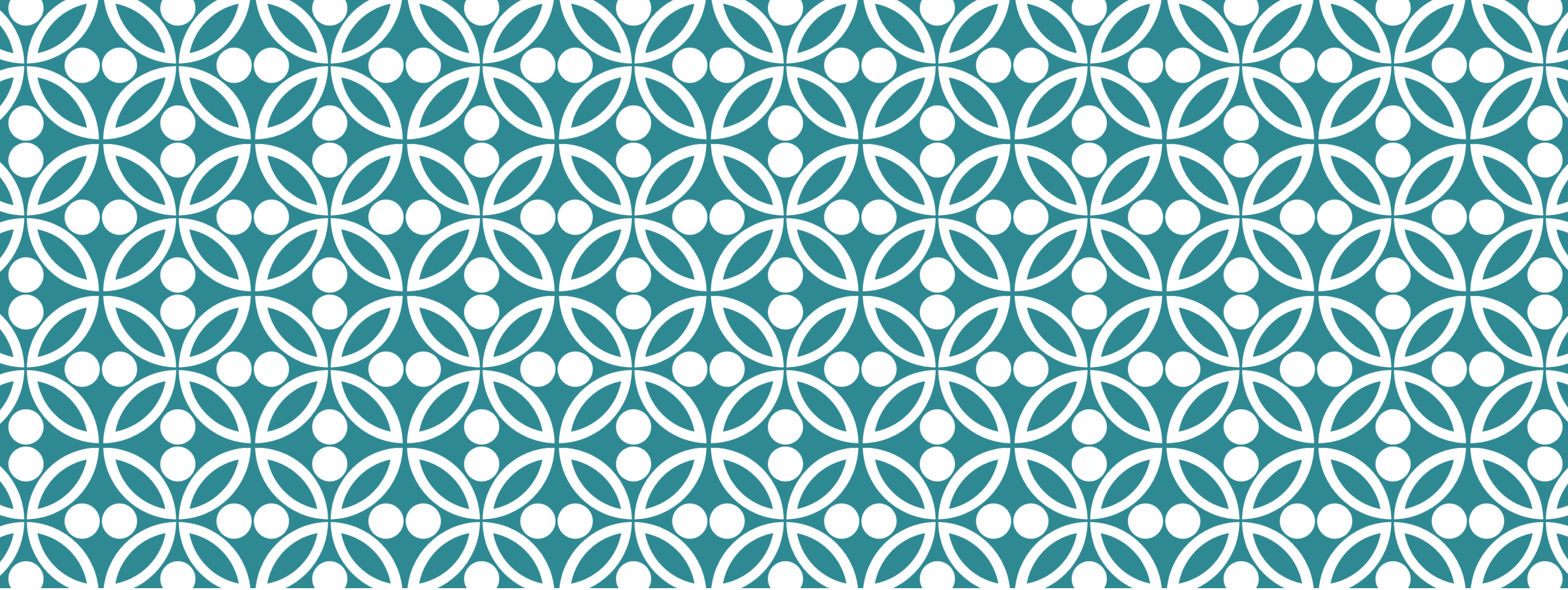

PROGRAMMING IN PYTHON

Gavrilut Dragos

Course 9

CLASSES

Classes exists in Python but have a different understanding about their functionality than the way classes are defined in C-like languages. Classes can be defined using a special keyword: **class**

Python 3.x

```
class <name>:
      <statement_1>
       …
      <statement_n>
```

Where **statement**$_i$ is usually a declaration of a method or data member.

Documentation for Python classes can be found on:

- Python 3: https://docs.python.org/3/tutorial/classes.html

CLASSES

Classes have a special keyword (**self**) that resembles the keyword **this** from c-like languages.

Whenever you reference a data member (variable that belongs to a class) within the class definition the **self** keyword must be used.

Constructors can be defined by creating a "**__init__**" function. "__init__" function must have the first parameter **self**.

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p = Point()
print (p.x,p.y)
```

Output

```
0 0
```

Class Point has two members (x and y)

CLASSES

For a function defined within a class to be a method of that class it has to have the first parameter **self.**

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetX(self):
                return self.x

p = Point()
print (p.GetX())
```

Output

```
0
```

CLASSES

Defining a function within a class without having the first parameter **self** means that that function is a static function for that class.

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetY():
                return self.y

p = Point()
print (p.GetY())
```

Execution error (GetY is static)

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
        def GetY():
                print("Test")

Point.GetY()
```

Output

```
Python 3: will print "Test" on the screen
```

CLASSES

A data member can also be defined directly in the class definition. However, if mutable object are used the behavior is different (similar in terms of behavior to a static

Python 3.x

```
class Point:
        x = 0
        y = 0

p1 = Point()
p2 = Point()
p1.x = 10
p2.x = 20
print (p1.x,p2.x)
```

Output

```
10 20
```

Python 3.x

```
class Point:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Point()
p2 = Point()
p1.AddNumber(4)
p2.AddNumber(5)
print (p1.numbers)
print (p2.numbers)
```

Output

```
[1,2,3,4,5]
[1,2,3,4,5]
```

CLASSES

To avoid problems with mutable objects it is better to defined them in a constructor (__init__) function:

Python 3.x

```
class Point:
        def __init__(self):
                self.numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Point()
p2 = Point()
p1.AddNumber(4)
p2.AddNumber(5)
print (p1.numbers)
print (p2.numbers)
```

Output

```
[1,2,3,4]
[1,2,3,5]
```

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print (p1.x,p1.y,p1.z)
```

Output

```
0 0 10
```

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print (p1.x,p1.y,p2.z)
```

Error during runtine. “p2” does not have a data member “z” (only “p1” has a data member “z”)

CLASSES

It is not required for two instances of the same class to have the same members. A class instance is more like a dictionary where each key represent either a member function or a data member

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
print ("x" in dir(p1))
print ("z" in dir(p1))
print ("z" in dir(p2))
```

Output

```
True
True
False
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0

p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

We can write an equivalent representation of the functionality done by classes by using dictionaries:

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p1 = Point()
p2 = Point()
p1.z = 10
```

Python 3.x (dictionary representation)

```
def PointClass__init__(obj):
        obj["x"] = 0
        obj["y"] = 0

Point = { "__init__":PointClass__init__ }
p1 = dict(Point)
p1["__init__"](p1)
p2 = dict(Point)
p2["__init__"](p2)
p1["z"] = 10
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

What happens if a class has some objects defined directly in class ?

Python 3.x

```
class Test:
        numbers = [1,2,3]
        def AddNumber(self,n):
                self.numbers += [n]
p1 = Test()
p2 = Test()
p1.AddNumber(4)
p2.AddNumber(5)
```

As both **p1.numbers** and **p2.numbers** refer to the same vector (**numbers_vector**) they will both modify the same object thus creating the illusion of a static variable.

Python 3.x (dictionary representation)

```
numbers_vector = [1,2,3]
def TestClass_AddNumber(obj,n):
        obj["numbers"]+=[n]

TestClass = {
    "AddNumber":TestClass_AddNumber,
    "numbers":numbers_vector
}

p1 = dict(TestClass)
p2 = dict(TestClass)
p1["AddNumber"](p1,4)
p2["AddNumber"](p2,5)
```

CLASSES

You can also delete a member of a class instance by using the keyword **del.**

Python 3.x

```
class Point:
        def __init__(self):
                self.x = 0
                self.y = 0
p = Point()
print (p.x,p.y)
p.x = 10
print (p.x,p.y)
del p.x
print (p.x,p.y)
```

"x" is no longer a member of p. Code will produce a runtine error.

CLASSES

If a class member is like a dictionary – what does this means in terms of POO concepts:

A. method overloading is NOT possible (it would mean to have multiple functions with the same key in a dictionary). You can however create one method with a lot of parameters with default values that can be used in the same way.

B. There are no private/protected attributes for data members in Python. This is not directly related to the similarity to a dictionary, but it is easier this way as all keys from a dictionary are accessible.

C. CAST-ing does not work in the same way as expected. Up-cast / Down-cast are usually done with specialized functions that create a new object

D. Polymorphism is implicit (basically all you need to have is some classes with some functions with the same name). Even if this supersedes the concept of polymorphism, you don't actually need to have classes that are derived from the same class to simulate a polymorphism mechanism.

CLASSES

Just like normal variables in Python, data members can also have their type changed dynamically.

Python 3.x

```
class MyClass:
        x = 10
        y = 20

m = MyClass()
print (m.x,"=>",type(m.x))
m.x = "a string"
print (m.x,"=>",type(m.x))
```

Output

```
10 => <class 'int'>
a string => <class 'str'>
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = m.MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2
                                                    lf.y)
m = MyClass()
print (m.Test(15),m.Test(
m.Test = MyClass.MyFunction
print (m.Test(1,2))
```

Runtime error because “MyFunction” is a method that needs to be bound to an object instance !

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = MyClass().MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

The same can be applied for class methods – however in this case there are some restrictions related to the **self** keyword.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
m2 = MyClass()
print (m.Test(15),m.Test(16))
m.Test = m2.MyFunction
print (m.Test(1,2))
```

Output

```
True False
3 - 10,20
```

CLASSES

Methods are bound to the **self** object of the class they were initialized in. Even if you associate a method from a different class to a new method, the **self** will belong to the original class.

Python 3.x

```
class MyClass:
        x = 10
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)
m = MyClass()
m2 = MyClass()
m2.x = 100
m.Test = m2.MyFunction
print (m.Test(1,2))
print (m.MyFunction(1,2))
```

m.Test actually refers to **m2.MyFunction**

Output

```
3 - 100
3 - 10
```

CLASSES

A method from another class can also be used, but it will refer to the self from the

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
class AnotherClass:
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - "+str(self.x)+","+str(self.y)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = AnotherClass().MyFunction
print (m.Test(1,2))
```

The code will produce a runtime error because the **self** object from AnotherClass does not have “x” and “y” members.

CLASSES

Normal functions can also be used. However, in this case, the **self** object will not be send when calling them and it will not be accessible.

Python 3.x

```
class MyClass:
        x = 10
        y = 20
        def Test(self,value):
                return ((self.x+self.y)/2 == value)
def MyFunction(self,v1,v2):
        return str(v1+v2)
m = MyClass()
print (m.Test(15),m.Test(16))
m.Test = MyFunction
print (m.Test(1,2))
```

Output

```
True False
3
```

CLASSES

Similarly a class method can be associated (linked) to a normal variable and used as such. It will be able to use the **self** and it will be affected if **self** members are changed.

Python 3.x

```
class MyClass:
        x = 10
        def MyFunction(self,v1,v2):
                return str(v1+v2)+" - self.x:"+str(self.x)
m = MyClass()
fnc = m.MyFunction
print (fnc(15,35))
m.x = 123
print (fnc(15,35))
```

Output

```
50 - self.x: 10
50 - self.x: 123
```

CLASSES

self object is assign during the construction of an object. This means that a function can be defined outside the class and used within the class if it is set during the construction phase.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10
        Test = MyFunction

m = MyClass()
m2 = MyClass()
m2.x = 15
print (m.Test(1,2))
print (m2.Test(10,20))
```

Output

```
3 - X = 10
30 - X = 15
```

CLASSES

This type of assignment can not be done within the constructor method (__init__), it must be done through direct declaration in the class body.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10
        def __init__(self):
                self.Test = MyFunction
m = MyClass()
m2 = MyClass()
m2.x = 15
print (m.Test(1,2))
print (m2.Test(10,20))
```

The code will produce a runtime error because MyFunction is not bound to any **self** at this point

CLASSES

The same error will appear if we try to link a method from a class using it's instance with a non-class function.

Python 3.x

```
def MyFunction(self,v1,v2):
        return str(v1+v2)+" - X = "+str(self.x)

class MyClass:
        x = 10

m = MyClass()
m2.Test = MyFunction

print (m.Test(1,2))
```

The code will produce a runtime error because MyFunction is not bound to any **self** at this point

CLASSES

A class can be used like a container of data (a sort of name dictionary). It's closest resemblance is to a **struct** in C-like languages. For this an empty class need to be create (using keyword **pass**)

Python 3.x

```
class Point:
        pass

p = Point()
p.x = 100
p.y = 200
p_3d = Point()
p_3d.x = 10
p_3d.y = 20
p_3d.z = 30
print ("P = ",p.x,p.y)
print ("3D= ",p_3d.x,p_3d.y,p_3d.z)
```

Output

```
P =  100 200
3D=  10 20 30
```