Object Oriented Programming

#10

Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Simple programming tasks are easily implemented as simple functions, but as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage. As functions become too large, you might break them into smaller functions and pass data from one to the other.

However, as the number of functions becomes large, designing and managing the data passed to functions becomes difficult and error prone.

At this point, you should consider moving your programming tasks to **object-oriented designs**.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a style of programming that focuses on an application's data and the methods you need to manipulate that data.

Object-oriented programming uses all of the concepts you are familiar with from modular procedural programming, such as variables, modules, and passing values to modules. Modules in object-oriented programs continue to use sequence, selection, and looping structures and make use of arrays.

However, object-oriented programming adds several new concepts to programming and involves **a different way of thinking**.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

In object-oriented terminology, an object is one concrete example of a class, and a class is a term that describes a group of objects with common properties. A class definition describes what attributes its objects will have and what those objects will be able to do. In other words, a class definition describes data and methods.

For example, Automobile is a class of objects. Automobile objects contain data or attributes such as a make, model, year, and color. Automobile objects also have access to methods such as going forward, going in reverse, and being filled with gasoline. An instance of a class is an existing object of a class. For example, my car is one instance of the Automobile class and my neighbor's car is another.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# Procedural light switch
def turnOn():
 global switchIsOn
 # turn the light on
 switchIsOn = True

def turnOff():
 global switchIsOn
 # turn the light off
 switchIsOn = False

# Main code
switchIsOn = False # a global Boolean variable
```

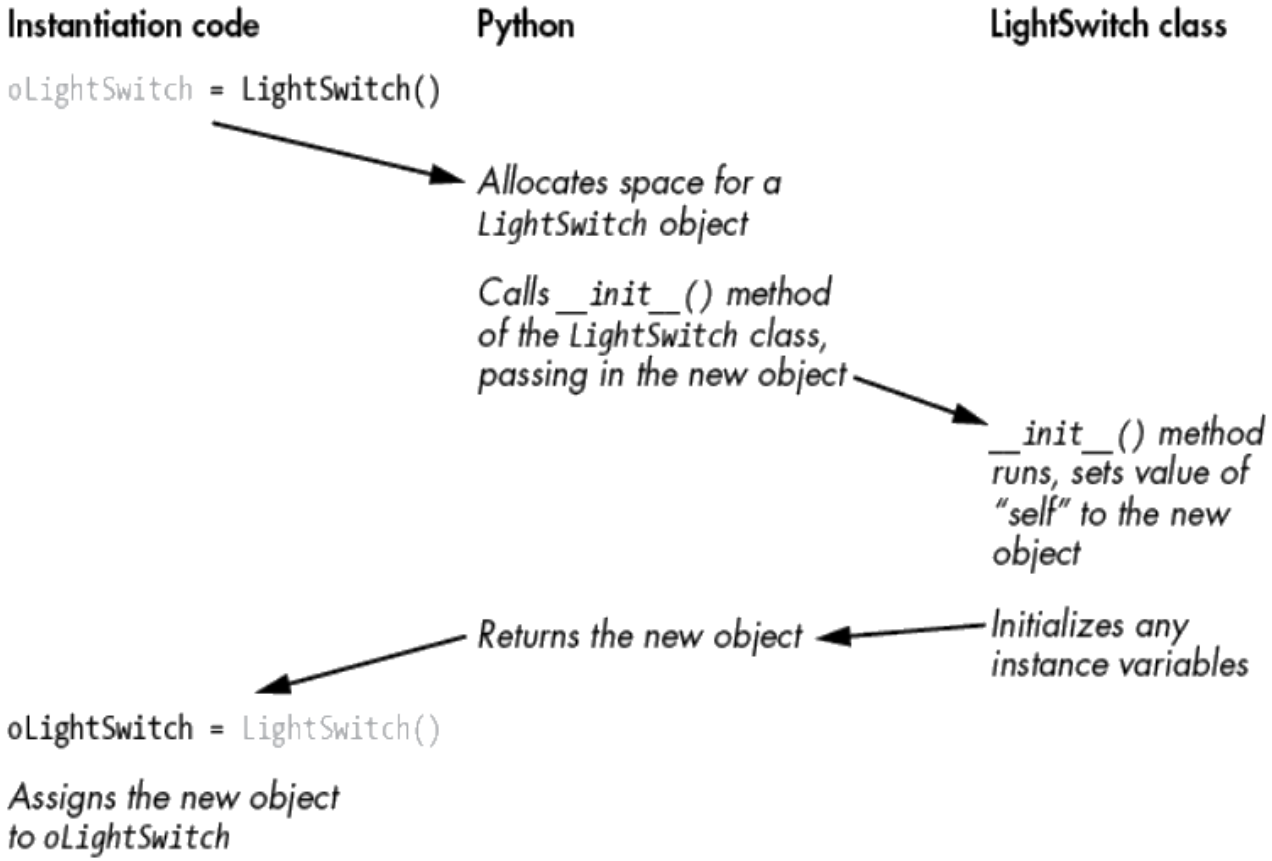
```
# Test code
print(switchIsOn)
turnOn()
print(switchIsOn)
turnOff()
print(switchIsOn)
turnOn()
print(switchIsOn)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# OO light switch
class LightSwitch():
    def __init__(self):
        self.switchIsOn = False

    def turnOn(self):
    # turn the switch on
        self.switchIsOn = True

    def turnOff(self):
    # turn the switch off
        self.switchIsOn = False

    def show(self): # added for testing
        print(self.switchIsOn)
```

```
# Test code
# create a LightSwitch object
oLightSwitch = LightSwitch()
# Calls to methods
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
oLightSwitch.turnOff()
oLightSwitch.show()
oLightSwitch.turnOn()
oLightSwitch.show()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING



CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
# OO light switch
# Main code

oLightSwitch1 = LightSwitch() # create a LightSwitch object
oLightSwitch2 = LightSwitch() # create another LightSwitch object

# Test code
oLightSwitch1.show()
oLightSwitch2.show()
oLightSwitch1.turnOn() # Turn switch 1 on
# Switch 2 should be off at start, but this makes it clearer
oLightSwitch2.turnOff()
oLightSwitch1.show()
oLightSwitch2.show()
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

objbrowser 1.3.1

```
pip install objbrowser
```

Latest version

Released: Dec 18, 2022

GUI for Python object introspection.

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage

Statistics

GitHub statistics:

- Stars: 117

Project description

Extensible Python object inspection tool implemented in Qt.

Displays objects as trees and allows you to inspect their attributes recursively (e.g. browse through a list of dictionaries). You can add your own inspection methods as new columns to the tree view, or as radio buttons to the details pane. Altering existing inspection methods is possible as well.

Requires: PySide or PyQt5

Installation: pip install objbrowser

Example use:

```
from objbrowser import browse
a = 16; b = 'hello'
browse(locals())
```

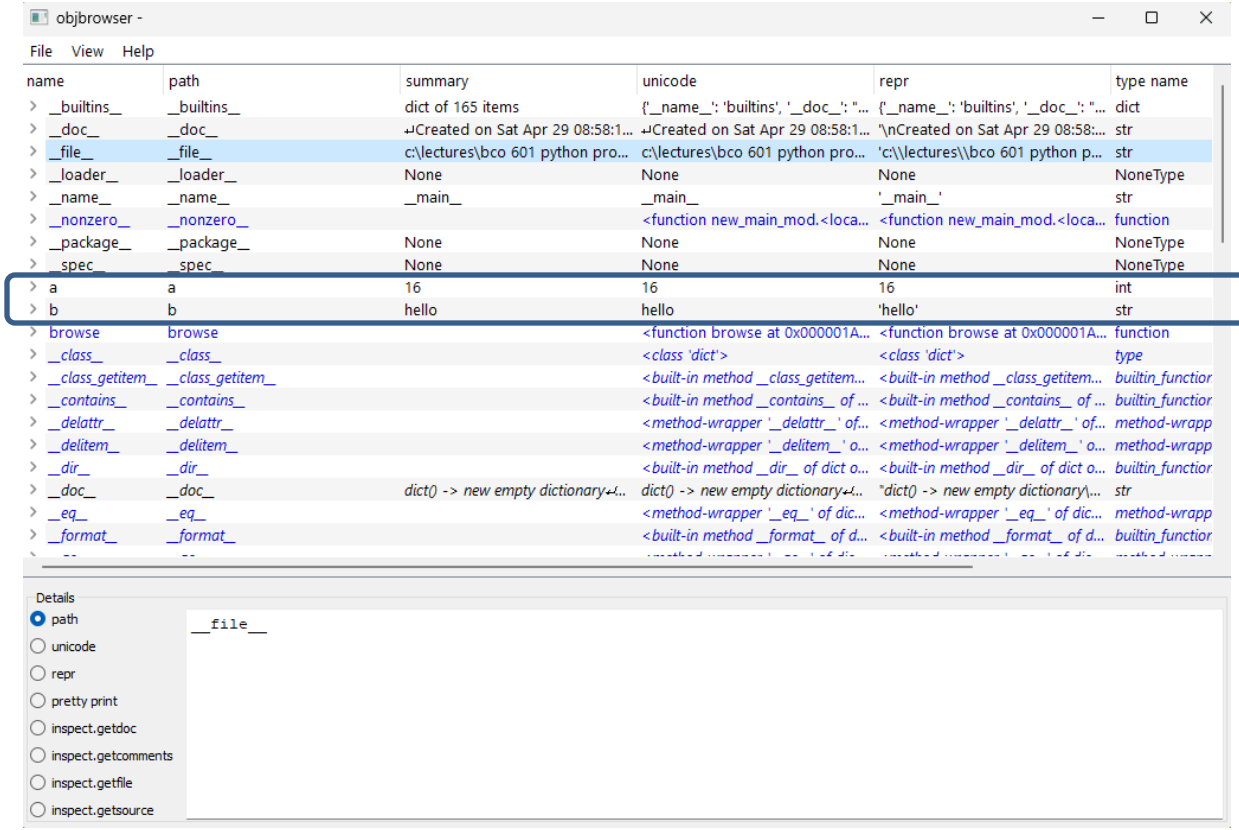
For more examples see: https://github.com/titusjan/objbrowser

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
from objbrowser import browse

a = 16
b = 'hello'
browse(locals())
```

PyQt5 5.15.11

```
pip install PyQt5
```



CLASSES AND OBJECT-ORIENTED PROGRAMMING

name	path	summary	unicode	repr	type name
__spec__	__spec__	None	None	None	NoneType
a	a	16	16	16	int
__abs__	a.__abs__		<method-wrapper '__abs__' of in...	<method-wrapper '__abs__' of in...	method-wrapp
__add__	a.__add__		<method-wrapper '__add__' of in...	<method-wrapper '__add__' of in...	method-wrapp
__and__	a.__and__		<method-wrapper '__and__' of in...	<method-wrapper '__and__' of in...	method-wrapp
__bool__	a.__bool__		<method-wrapper '__bool__' of i...	<method-wrapper '__bool__' of i...	method-wrapp
__ceil__	a.__ceil__		<built-in method __ceil__ of int o...	<built-in method __ceil__ of int o...	builtin_functior
__class__	a.__class__		<class 'int'>	<class 'int'>	type
__delattr__	a.__delattr__		<method-wrapper '__delattr__' of...	<method-wrapper '__delattr__' of...	method-wrapp
__dir__	a.__dir__		<built-in method __dir__ of int o...	<built-in method __dir__ of int o...	builtin_functior
__divmod__	a.__divmod__		<method-wrapper '__divmod__' ...	<method-wrapper '__divmod__' ...	method-wrapp
__doc__	a.__doc__	int([x]) -> integer↵int(x, base=1...	int([x]) -> integer↵int(x, base=1...	"int([x]) -> integer\nint(x, base=...	str
__eq__	a.__eq__		<method-wrapper '__eq__' of int ...	<method-wrapper '__eq__' of int ...	method-wrapp
__float__	a.__float__		<method-wrapper '__float__' of i...	<method-wrapper '__float__' of i...	method-wrapp
__floor__	a.__floor__		<built-in method __floor__ of int ...	<built-in method __floor__ of int ...	builtin_functior
__floordiv__	a.__floordiv__		<method-wrapper '__floordiv__' ...	<method-wrapper '__floordiv__' ...	method-wrapp
__format__	a.__format__		<built-in method __format__ of i...	<built-in method __format__ of i...	builtin_functior
__ge__	a.__ge__		<method-wrapper '__ge__' of int ...	<method-wrapper '__ge__' of int ...	method-wrapp
__getattrib...	a.__getattribute__		<method-wrapper '__getattribute...	<method-wrapper '__getattribute...	method-wrapp
__getnewar...	a.__getnewargs__		<built-in method __getnewargs__...	<built-in method __getnewargs__...	builtin_functior

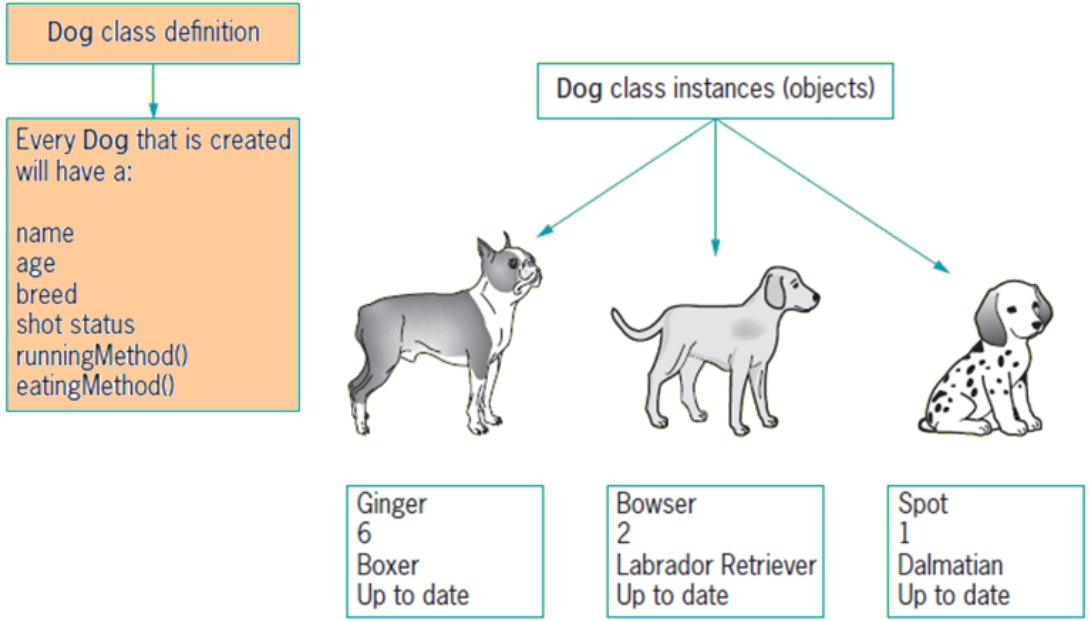
Details

path

__file__

CLASSES AND OBJECT-ORIENTED PROGRAMMING

When you program in object-oriented languages, you frequently create classes from which objects will be instantiated. Creating an object is called **instantiating** it.



CLASSES AND OBJECT-ORIENTED PROGRAMMING

A class in Python is effectively a data type. All the data types built into Python are classes, and Python gives you powerful tools to manipulate every aspect of a class's behavior. You define a class with the class statement:

```
class MyClass:
    body
```

body is a list of Python statements, typically variable assignments and function definitions. No assignments or function definitions are required. The body can be just a single `pass` statement.

By convention, `class` identifiers are in CapCase — that is, the first letter of each component word is capitalized, to make them stand out.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Class instances can be used as structures or records. Unlike C structures, the fields of an instance don't need to be declared ahead of time but can be created on the fly. The following short example defines a class called Circle, creates a Circle instance, assigns to the radius field of the circle, and then uses that field to calculate the circumference of the circle:

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> print('CemberinAlanı:',3.14*my_class.radius**2))
78.5
```

Page 15
Page 16

CLASSES AND OBJECT-ORIENTED PROGRAMMING

File View Help

name	path	summary	unicode	repr	type name	is callable
__class__	__class__		<class '__main__.Circle'>	<class '__main__.Circle'>	type	True
__delattr__	__delattr__		<method-wrapper '__delattr__' of...	<method-wrapper '__delattr__' of...	method-wrapper	True
__dict__	__dict__	empty dict	{}	{}	dict	False
__dir__	__dir__		<built-in method __dir__ of Circle...	<built-in method __dir__ of Circle...	builtin_function_or_method	True
__doc__	__doc__	None	None	None	NoneType	False
__eq__	__eq__		<method-wrapper '__eq__' of Cir...	<method-wrapper '__eq__' of Cir...	method-wrapper	True
__format__	__format__		<built-in method __format__ of C...	<built-in method __format__ of C...	builtin_function_or_method	True
__ge__	__ge__		<method-wrapper '__ge__' of Cir...	<method-wrapper '__ge__' of Cir...	method-wrapper	True
__getattribute__	__getattribute__		<method-wrapper '__getattribute...	<method-wrapper '__getattribute...	method-wrapper	True
__getstate__	__getstate__		<built-in method __getstate__ of ...	<built-in method __getstate__ of ...	builtin_function_or_method	True
__gt__	__gt__		<method-wrapper '__gt__' of Cir...	<method-wrapper '__gt__' of Cir...	method-wrapper	True
__hash__	__hash__		<method-wrapper '__hash__' of ...	<method-wrapper '__hash__' of ...	method-wrapper	True
__init__	__init__		<method-wrapper '__init__' of Ci...	<method-wrapper '__init__' of Ci...	method-wrapper	True
__init_subclass__	__init_subclass__		<built-in method __init_subclass_...	<built-in method __init_subclass_...	builtin_function_or_method	True
__le__	__le__		<method-wrapper '__le__' of Circ...	<method-wrapper '__le__' of Circ...	method-wrapper	True
__lt__	__lt__		<method-wrapper '__lt__' of Circl...	<method-wrapper '__lt__' of Circl...	method-wrapper	True
__module__	__module__	__main__	__main__	'__main__'	str	False
__ne__	__ne__		<method-wrapper '__ne__' of Cir...	<method-wrapper '__ne__' of Cir...	method-wrapper	True
__new__	__new__		<built-in method __new__ of type...	<built-in method __new__ of type...	builtin_function_or_method	True
__reduce__	__reduce__		<built-in method __reduce__ of C...	<built-in method __reduce__ of C...	builtin_function_or_method	True
__reduce_ex__	__reduce_ex__		<built-in method __reduce_ex__ o...	<built-in method __reduce_ex__ o...	builtin_function_or_method	True
__repr__	__repr__		<method-wrapper '__repr__' of C...	<method-wrapper '__repr__' of C...	method-wrapper	True
__setattr__	__setattr__		<method-wrapper '__setattr__' of...	<method-wrapper '__setattr__' of...	method-wrapper	True
__sizeof__	__sizeof__		<built-in method __sizeof__ of Ci...	<built-in method __sizeof__ of Ci...	builtin_function_or_method	True
__str__	__str__		<method-wrapper '__str__' of Cir...	<method-wrapper '__str__' of Cir...	method-wrapper	True
__subclasshook__	__subclasshook__		<built-in method __subclasshook...	<built-in method __subclasshook...	builtin_function_or_method	True
__weakref__	__weakref__	None	None	None	NoneType	False

CLASSES AND OBJECT-ORIENTED PROGRAMMING

name	path	summary	unicode	repr	type name	is callable
__class__	__class__		<class '__main__.Circle'>	<class '__main__.Circle'>	type	True
__delattr__	__delattr__		<method-wrapper '__delattr__' of...	<method-wrapper '__delattr__' of...	method-wrapper	True
__dict__	__dict__	dict of 1 item	{'radius': 5}	{'radius': 5}	dict	False
__dir__	__dir__		<built-in method __dir__ of Circle...	<built-in method __dir__ of Circle...	builtin_function_or_method	True
__doc__	__doc__	None	None	None	NoneType	False
__eq__	__eq__		<method-wrapper '__eq__' of Cir...	<method-wrapper '__eq__' of Cir...	method-wrapper	True
__format__	__format__		<built-in method __format__ of C...	<built-in method __format__ of C...	builtin_function_or_method	True
__ge__	__ge__		<method-wrapper '__ge__' of Cir...	<method-wrapper '__ge__' of Cir...	method-wrapper	True
__getattribute__	__getattribute__		<method-wrapper '__getattribute...	<method-wrapper '__getattribute...	method-wrapper	True
__getstate__	__getstate__		<built-in method __getstate__ of ...	<built-in method __getstate__ of ...	builtin_function_or_method	True
__gt__	__gt__		<method-wrapper '__gt__' of Cir...	<method-wrapper '__gt__' of Cir...	method-wrapper	True
__hash__	__hash__		<method-wrapper '__hash__' of ...	<method-wrapper '__hash__' of ...	method-wrapper	True
__init__	__init__		<method-wrapper '__init__' of Ci...	<method-wrapper '__init__' of Ci...	method-wrapper	True
__init_subclass__	__init_subclass__		<built-in method __init_subclass_...	<built-in method __init_subclass_...	builtin_function_or_method	True
__le__	__le__		<method-wrapper '__le__' of Circ...	<method-wrapper '__le__' of Circ...	method-wrapper	True
__lt__	__lt__		<method-wrapper '__lt__' of Circl...	<method-wrapper '__lt__' of Circl...	method-wrapper	True
__module__	__module__	__main__	__main__	'__main__'	str	False
__ne__	__ne__		<method-wrapper '__ne__' of Cir...	<method-wrapper '__ne__' of Cir...	method-wrapper	True
__new__	__new__		<built-in method __new__ of type...	<built-in method __new__ of type...	builtin_function_or_method	True
__reduce__	__reduce__		<built-in method __reduce__ of C...	<built-in method __reduce__ of C...	builtin_function_or_method	True
__reduce_ex__	__reduce_ex__		<built-in method __reduce_ex__ o...	<built-in method __reduce_ex__ o...	builtin_function_or_method	True
__repr__	__repr__		<method-wrapper '__repr__' of C...	<method-wrapper '__repr__' of C...	method-wrapper	True
__setattr__	__setattr__		<method-wrapper '__setattr__' of...	<method-wrapper '__setattr__' of...	method-wrapper	True
__sizeof__	__sizeof__		<built-in method __sizeof__ of Ci...	<built-in method __sizeof__ of Ci...	builtin_function_or_method	True
__str__	__str__		<method-wrapper '__str__' of Cir...	<method-wrapper '__str__' of Cir...	method-wrapper	True
__subclasshook__	__subclasshook__		<built-in method __subclasshook...	<built-in method __subclasshook...	builtin_function_or_method	True
__weakref__	__weakref__	None	None	None	NoneType	False
radius	radius	5	5	5	int	False

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Like many other languages, the fields of an instance / structure are accessed and assigned to by using dot notation. You can initialize fields of an instance automatically by including an `__init__` initialization method in the class body. This function is run every time an instance of the class is created, with that new instance as its first argument. The `__init__` method is similar to a constructor, but it doesn't really construct anything— it initializes fields of the class.

```
class Circle:
        def __init__(self):
                self.radius = 1

my_circle = Circle()
print(2 * 3.14 * my_circle.radius)
my_circle.radius = 5
print(2 * 3.14 * my_circle.radius)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Instance variables are the most basic feature of OOP. Take a look at the Circle class again:

```
class Circle:
        def __init__(self):
                self.radius = 1
```

radius is an instance variable of Circle instances. That is, each instance of the Circle class has its own copy of radius, and the value stored in that copy may be different from the values stored in the radius variable in other instances. In Python, you can create instance variables as necessary by assigning to a field of a class instance:

```
instance.variable = value
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A METHOD is a function associated with a particular class. You've already seen the special `__init__` method, which is called on a new instance when that instance is first created. In the following example, we define another method, area, for the Circle class, which can be used to calculate and return the area for any Circle instance. Like most user-defined methods, area is called with a method invocation syntax that resembles instance variable access:

```
class Circle:
    def __init__(self):
        self.radius = 1
    def area(self):
        return self.radius **2 * 3.14159
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Circle:
    def __init__(self):
        self.radius = 1
    def area(self):
        return self.radius **2 * 3.14159
```

Method invocation syntax consists of an instance, followed by a period, followed by the method to be invoked on the instance.

```
>>> my_circle = Circle()
>>> print(2 * 3.14 * my_circle.radius)
>>> my_circle.radius = 5
>>> print(2 * 3.14 * my_circle.radius)
>>> print(my_circle.area())
```

Write a Method that calculates the circumference of the circle

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Methods can be invoked with arguments, if the method definitions accept those arguments. This version of Circle adds an argument to the `__init__` method, so that we can create circles of a given radius without needing to set the radius after a circle is created:

```
class Circle:
        def __init__(self, radius):
                self.radius = radius
        def area(self):
                return self.radius ** 2 * 3.14159
```

Note the two uses of radius here. self.radius is the instance variable called radius. radius by itself is the local function variable called radius. The two aren't the same! In practice, we'd probably call the local function variable something like r or rad, to avoid any possibility of confusion.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

All the standard Python function features—default argument values, extra arguments, keyword arguments, and so forth—can be used with methods. For example, we could have defined the first line of `__init__` to be

```
def __init__(self, radius=1):
```

Then, calls to circle would work with or without an extra argument; `Circle()` would return a circle of radius 1, and `Circle(3)` would return a circle of radius 3.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

A **class variable** is a variable associated with a class, **not** an instance of a class, and is accessed by all instances of the class, in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time. A class variable is created by an assignment in the class body, not in the `__init__` function; after it has been created, it can be seen by all instances of the class.

```
class Circle:
        pi = 3.14159
        def __init__(self, radius):
                self.radius = radius
        def area(self):
                return self.radius **2 * Circle.pi
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Circle:
        pi = 3.14159
        def __init__(self, radius):
                self.radius = radius
        def area(self):
                return self.radius **2 * Circle.pi

>>> Circle.pi
3.14159
>>> Circle.pi = 4
>>> Circle.pi
4
>>> Circle.area()
Circle.area()
TypeError: area() missing 1 required positional argument: 'self'
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Circle:
        pi = 3.14159
        def __init__(self, radius):
                self.radius = radius
        def area(self):
                return self.radius **2 * Circle.pi
<< Run Module F5 >>
>>> Circle
<class '__main__.Circle'>
>>> c = Circle(3)
>>> c.__class__
<class '__main__.Circle'>

>>> from circle import *
>>> Circle
<class 'circle.Circle'>
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> Circle.pi
3.14159
>>> c1 = Circle(1)
>>> c2 = Circle(2)
>>> c1.pi
3.14159
>>> c2.pi
3.14159
>>> Circle.pi = 6.28
>>> Circle.pi
6.28
>>> c1.pi
6.28
>>> c2.pi
6.28
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
============== RESTART ==============
>>> c1 = Circle(1)
>>> c2 = Circle(2)
>>> c1.pi = 6.28
>>> c2.pi
3.14159
>>> Circle.pi
3.14159
>>> c1.pi
6.28
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

You may object to hardcoding the name of a class inside that class's methods. You can avoid doing so through use of the special `__class__` attribute, available to all Python class instances.

```
class Circle:
    pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius **2 * self.__class__.pi
```

```
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  4 2021, 13:27:16) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: C:\Lectures\BCO 601 Python Programming\OOP\circle.py ========
>>> Circle.
area
pi
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Leading Underscore before variable/function/method name indicates to programmer that It is for internal use only, that can be modified whenever class want.

```
class Circle:
    _pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius **2 * self.__class__._pi
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Python does not specify truly private so this ones can be call directly from other modules if it is specified in __all__, We also call it weak Private.

```
>>> dir(Circle)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_pi', 'area']
>>>
```

Underscore (_) in Python

In Interpreter:

_ returns the value of last executed expression value in Python Prompt/Interpreter

```
>>> a = 10
>>> b = 10
>>> _
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> a+b
20
>>> _
20
>>> _ * 2
40
>>> _
40
```

Underscore (_) in Python

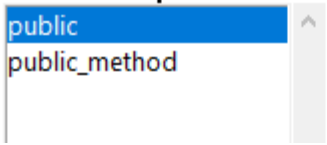
For ignoring values:
Multiple time we do not want return values at that time assign those values to Underscore. It used as throwaway variable.

```
# Ignore a value of specific location/index
for _ in range(10)
    print("Test")

# Ignore a value when unpacking
a,b,_,_ = my_method(var1)
```

Underscore (_) in Python

```
class MyPrivateClass():
    def __init__(self):
        self.public = 10
        self._private = 12

    def public_method(self):
        print ("public method")

    def _private_method(self):
        print ("private method")
```

```
>>> from myPrivateClass import *
>>> test = MyPrivateClass()
>>> test.public
```



Underscore (_) in Python

```
>>> from myPrivateClass import *
>>> test = MyPrivateClass()
>>> test.public
10
>>> test._private
12
>>> test.public_method()
public method
>>> test._private_method()
private method
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
class Circle:
        _pi = 3.14159
        def __init__(self, radius):
                self.radius = radius
        def area(self):
                return self.radius **2 * self.__class__._pi
<< Run Module F5 >>
>>> Circle
<class '__main__.Circle'>
>>> c = Circle(3)
>>> c.__class__
<class '__main__.Circle'>

>>> from circle import *
>>> Circle
<class 'circle.Circle'>
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
c = Circle()
TypeError: __init__() missing 1 required positional argument: 'radius'
```

```
class Circle:
    _pi = 3.14159
    def __init__(self, radius = 1):
        self.radius = radius
    def area(self):
        return self.radius **2 * self.__class__._pi
```

```
>>> c = Circle()
>>> c.radius
1
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    _pi = 3.14159

    def __init__(self, radius = 1):
        """Create a Circle with the given radius"""
        self.radius = radius
    def area(self):
        """determine the area of the Circle"""
        return self.radius **2 * self.__class__._pi

>>> import circle
>>> circle.Circle.__doc__
'Circle class'
>>> c = circle.Circle()
>>> c.__doc__
'Circle class'
>>> c.area.__doc__
'determine the area of the Circle'
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> import os
>>> os.getcwd()
'C:\\Program Files\\Python39'
>>> os.chdir('C:\Lectures\BCO 601 Python Programming\OOP')
>>> os.getcwd()
'C:\\Lectures\\BCO 601 Python Programming\\OOP'
>>> import circle
>>> dir(circle)
['Circle', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__']
>>> circle.Circle.area()
TypeError: area() missing 1 required positional argument: 'self'
>>> c1 = circle.Circle(1)
>>> c1.area()
3.14159
>>> c2 = circle.Circle(2)
>>> c2.area()
12.56636
>>> areaSum = c1.area() + c2.area()
>>> print(areaSum)
15.70795
```

Call but no instance yet!!!

c1 and c2 are new instances

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Static methods even though no instance of that class has been created, although you can call them using a class instance. To create a static method, use the `@staticmethod` decorator.

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    _all_circles = []
    _pi = 3.14159

    def __init__(self, r=1):
        """Create a Circle with the given radius"""
        self.radius = r
        self.__class__.all_circles.append(self)

    def area(self):
        """determine the area of the Circle"""
        return self.__class__._pi * self.radius ** 2

    @staticmethod
    def total_area():
        total = 0
        for c in Circle._all_circles:
            total = total + c.area()
        return total
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> import circle
circle.Circle.area()
TypeError: area() missing 1 required positional argument: 'self'
>>> circle.Circle.total_area()
0
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> c1.area()
3.14159
>>> c2.area()
12.56636
>>> c1.area() + c2.area()
15.70795
>>> circle.Circle.total_area()
15.70795
>>> c1.total_area()
15.70795
>>> c2.total_area()
15.70795
>>>
```

Call but no instance yet!!!

c1 and c2 are new instances

```
======= RESTART: C:\Lectures\BCO 601 Python Programming\OOP\test_pair.py =======
>>> import circle
>>> circle.Circle.total_area()
0
>>> circle.Circle.
area
total_area
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter, so you can code them more simply.

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    _all_circles = []
    _pi = 3.14159
            .
            .
    @classmethod
    def total_area(cls):
            total = 0
            for c in cls._all_circles:
                    total = total + c.area()
            return total
```

The class parameter is traditionally `cls`

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> import circle
>>> circle.Circle.total_area()
0
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> circle.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle_cm.Circle.total_area()
31.4159
>>> c1.total_area()
31.4159
>>> c2.total_area()
31.4159
```

By using a class method instead of a static method, we don't have to hardcode the class name into total_area. That means any subclasses of Circle can still call `total_area` and refer to their own members, not those in `Circle`.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
@staticmethod
def total_area():
        total = 0
        for c in Circle.all_circles:
                total = total + c.area()
        return total

@classmethod
def total_area(cls):
        total = 0
        for c in cls.all_circles:
                total = total + c.area()
        return total
```

By using a class method instead of a static method, we don't have to hardcode the class name into `total_area`.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

@classmethod and @staticmethod

```
from datetime import date

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a class method to create a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

@classmethod and @staticmethod

```
person1 = Person('Serdar', 58)
person2 = Person.fromBirthYear('Serdar', 1966)

print(person1.age)
print(person2.age)

# print the result
print(Person.isAdult(58))
```

```
58
58
True
```

- A class method takes `cls` as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can't access or modify it.
- In general, static methods know nothing about class state.
- They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
    # A class method can access or modify class state
    # while a static method can't access or modify it.
    @staticmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

Traceback (most recent call last):
person2 = Person.fromBirthYear('Serdar', 1966)
TypeError: fromBirthYear() missing 1 required positional argument: 'year'
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Changing the String Representation of Instances

`__str__()`

This method returns the string representation of the object. This method is called when `print()` or `str()` function is invoked on an object. This method must return the String object.

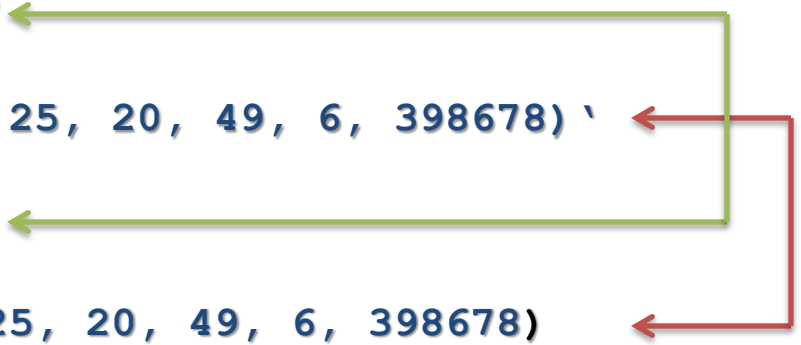
`__repr__()`

This method returns the object representation in string format. This method is called when `repr()` function is invoked on the object.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Changing the String Representation of Instances

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.__str__()
'2021-04-25 20:49:06.398678'
>>> now.__repr__()
'datetime.datetime(2021, 4, 25, 20, 49, 6, 398678)'
>>> print(now)
2021-04-25 20:49:06.398678
>>> now
datetime.datetime(2021, 4, 25, 20, 49, 6, 398678)
```



CLASSES AND OBJECT-ORIENTED PROGRAMMING

Changing the String Representation of Instances

To change the string representation of an instance, define the `__str__()` and `__repr__()` methods. For example:

```
class Pair:
        def __init__(self, x, y):
                self.x = x
                self.y = y
        def __repr__(self):
                return 'Pair({0.x!r}, {0.y!r})'.format(self)
        def __str__(self):
                return '({0.x!s}, {0.y!s})'.format(self)
```

The `__repr__()` method returns the code representation of an instance, and is usually the text you would type to re-create the instance. The `__str__()` method converts the instance to a string, and is the output produced by the `str()` and `print()` functions.

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # __repr__() output
>>> print(p)
(3, 4)              # __str__() output
>>>
```

`!r` calls `repr()` (which calls `__repr__` internally) on the object to get a string. Specifically, the special `!r` formatting code indicates that the output of `__repr__()` should be used instead of `__str__()`, the default. You can try this experiment with the preceding class to see this:

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

The explicit conversion flag is used to transform the format field value before it is formatted. This can be used to override the type-specific formatting behavior, and format the value as if it were a more generic type.

Currently, two explicit conversion flags are recognized.

```
!r - convert the value to a string using repr().
!s - convert the value to a string using str().

>>> "{0!r:20}".format("Hello")
"'Hello'             "
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Python's built-in `format(value, spec)` function transforms input of one format into output of another format defined by you..

```
_formats = {
            'row' : '{d.x}-{d.y}',
            'col' : '\n{d.x}\n{d.y}'
            }
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        #return 'Pair({0.x!r}, {0.y!r})'.format(self)
        #return 'Pair(%r, %r)' % (self.x, self.y)
        return  f'Pair({self.x}, {self.y})'

    def __str__(self):
        #return '({0.x!s}, {0.y!s})'.format(self)
        #return '(%s, %s)' % (self.x, self.y)
        return  f'({self.x}, {self.y})'

    def __format__(self, code):
        if code == '':
            code = 'row'
        fmt = _formats[code]
        return fmt.format(d=self)
```

```
>>> print(p)
(3, 4)

>>> print(format(p))
3-4

>>> print(format(p, 'col'))

3
4
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
"""circle module: contains the Circle class."""
class Circle:
        """Circle class"""
        _all_circles = []
        from math import pi
        def __init__(self, r=1):
                """Create a Circle with the given radius"""
                self.radius = r
                self.__class__._all_circles.append(self)
        def __str__(self):
                return ' r: ' + str(self.radius)
        def __repr__(self):
                return 'Circle(r = {0.radius!r})'.format(self)
        def area(self):
                """determine the area of the Circle"""
                return self.__class__.pi * self.radius **2
        @classmethod
        def total_area(cls):
                total = 0
                for c in cls._all_circles:
                        total = total + c.area()
                return total
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> c1 = Circle()
>>> c2 = Circle(2)
>>> c1
Circle(r = 1)
>>> c2
Circle(r = 2)
>>> print(c2)
 r: 2
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Operator overloading means that the operation performed by the operator depends on the type of operands provided to the operator.

Binary Operators	
Operator	**Method**
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__div__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Whenever the meaning of an operator is not obviously clear and undisputed, **it should not be overloaded**. Instead, provide a function with a well-chosen name.

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Always stick to the operator's well-known semantics.

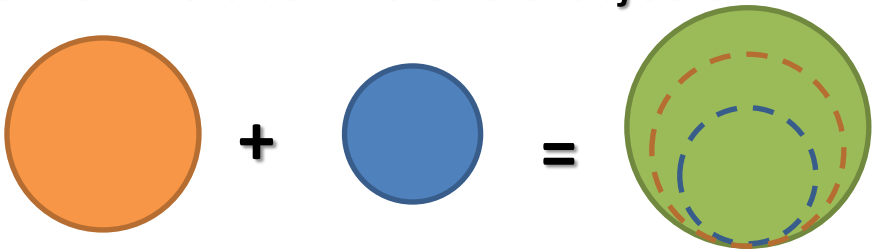
Unary Operators	
Operator	**Method**
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

CLASSES AND OBJECT-ORIENTED PROGRAMMING

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

If we want to add two circle objects!



```
object.__add__(self, other)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    all_circles = []
    from math import pi

    def __init__(self, r=1):
      """Create a Circle with the given radius"""
      self.radius = r
      self.__class__.all_circles.append(self)

    def __add__(self, other):
      if type(other) is  Circle:
        return Circle(self.radius + other.radius )
      else:
        print('Unsupported object for adding: Both object must be Circle')
```

```
>>> c1 = Circle(5)
>>> c2 = Circle(8)
>>> c3 = c1 + c2
>>> c3 + 5
>>> 5 + c3
```

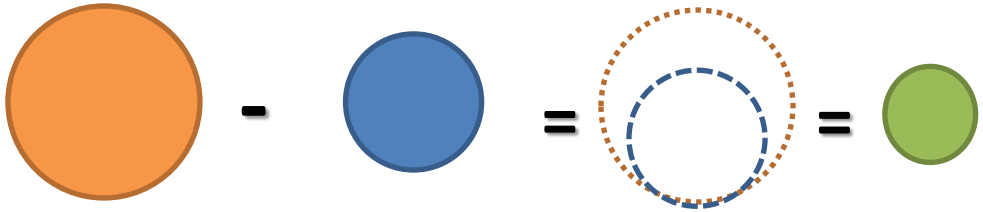
```
__add__(self, other):
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

```
>>> 5 + c3
```

```
def __radd__(self, other):
      if type(other) is  Circle:
        return Circle(self.radius + other.radius )
      else:
        print('Unsupported object for adding: Both object must be Circle')
```

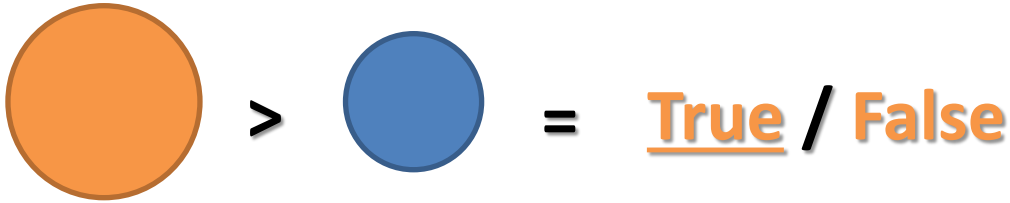
If we want to subtract two circle objects!



```
object.__sub__(self, other)
```

CLASSES AND OBJECT-ORIENTED PROGRAMMING

If we want to compare two circle objects!



= True / False

Comparison Operators

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

CLASSES AND OBJECT-ORIENTED PROGRAMMING

- Think before you program!
- A program is a human-readable essay on problem solving that also happens to execute on a computer.
- The best way to improve your programming and problem skills is to practice!
- Test your code, often and thoroughly!
- If it was hard to write, it is probably hard to read. Add a comment.
- All input is evil, until proven otherwise.
- A function should do one thing.

CLASSWORK

Write a simple interactive calculator that takes commands from the user. The calculator maintains and displays an Accumulator, which is the current stored value. Commands can be any of the following:

```
clear       zero the accumulator
show        display the accumulator value
add k       add k to the accumulator
sub k       subtract k from the accumulator
mult k      multiply accumulator by k
div k       divide accumulator by k
help        show commands
exit        terminate the computation
```

This version need only accept non-negative integer arguments, though it can yield negative and float results.

CLASSWORK

```
class Calc:
    """This is a simple calculator class. It stores and displays
    a single number in the accumulator. To that number, you can
    add, subtract, multiply or divide.
    """
    def __init__(self):
        "Constructor for new Calc objects, with display 0."
        self._accumulator = 0

    def getAccumulator(self):
        """Accessor for the accumulator."""
        return self._accumulator

    def __str__(self):
        """Printing displays the accumulator value."""
        return "Value is: " + str(self._accumulator)

    def clear(self):
        """Zero the accumulator."""
        self._accumulator = 0
        print(self)

    def add(self, num):
        """Add num to accumuator."""
        self._accumulator += num
        print(self)
```

CLASSWORK

```
    def sub(self, num):
        """Subtract num from accumuator."""

    def mult(self, num):
        """Multiply accumuator by num."""

    def div(self, num):
        """Divide accumuator by num (unless num is 0)."""


c = Calc()  # create a new calculator object
c.add(5)    # add 5 to the accumulator
c.add(3)    # add 3 to the accumulator
print(c)    # print the result
```

CLASSWORK

Let's consider a more complicated physical object: a television. With this more complicated example, we'll take a closer look at how arguments work in classes.

A simplified TV remote

- Power state (on or off)
- Mute state (is it muted?)
- List of channels available
- Current channel setting
- Current volume setting
- Range of volume levels available

And the actions that the TV must provide include:

- Turn the power on and off
- Raise and lower the volume
- Change the channel up and down
- Mute and unmute the sound
- Get information about the current settings
- Go to a specified channel

CLASSWORK

```
class TV():
    def __init__(self):
        self.isOn = False
        self.isMuted = False
        # Some default list of channels
        self.channelList = [2, 4, 5, 7, 9, 11, 20, 36, 44, 54, 65]
        self.nChannels = len(self.channelList)
        self.channelIndex = 0
        self.VOLUME_MINIMUM = 0 # constant
        self.VOLUME_MAXIMUM = 10 # constant
        self.volume = self.VOLUME_MAXIMUM

    def power(self): 2
        self.isOn = not self.isOn # toggle
```

CLASSWORK

```
def volumeUp(self):
    if not self.isOn:
        return
    if self.isMuted:
        self.isMuted = False # changing the volume while muted unmutes
    if self.volume < self.VOLUME_MAXIMUM:
        self.volume = self.volume + 1

def volumeDown(self):
    …
    …
    …
```

CLASSWORK

```
def channelUp(self):
    if not self.isOn:
        return
    self.channelIndex = self.channelIndex + 1
    if self.channelIndex > self.nChannels:
        self.channelIndex = 0 # wrap around to the first channel

def channelDown(self):
    …
    …
    …
```

CLASSWORK

```
def setChannel(self, newChannel):
    if newChannel in self.channelList:
    self.channelIndex = self.channelList.index(newChannel)
    # if the newChannel is not in our list of channels, don't do anything

 def mute(self):
      …
      …
      …
 def showInfo(self):
      …
      …
      …
```

CLASSWORK

```
# Main code
oTV = TV() # create the TV object
# Turn the TV on and show the status
oTV.power()
oTV.showInfo()
# Change the channel up twice, raise the volume twice, show status
oTV.channelUp()
oTV.channelUp()
oTV.volumeUp()
oTV.volumeUp()
oTV.showInfo()
# Turn the TV off, show status, turn the TV on, show status
oTV.power()
oTV.showInfo()
oTV.power()
oTV.showInfo()
# Lower the volume, mute the sound, show status
oTV.volumeDown()
oTV.mute()
oTV.showInfo()
# Change the channel to 11, mute the sound, show status
oTV.setChannel(11)
oTV.mute()
oTV.showInfo()
```

CLASSWORK

```
TV Status:
 TV is: On
 Channel is: 2
 Volume is: 5
TV Status:
 TV is: On
 Channel is: 5
 Volume is: 7
TV Status:
 TV is: Off
TV Status:
 TV is: On
 Channel is: 5
 Volume is: 7
TV Status:
 TV is: On
 Channel is: 5
 Volume is: 6 (sound is muted)
TV Status:
 TV is: On
 Channel is: 11
 Volume is: 6
```