# Object Oriented Programming: Python's Take

OOP is a programming paradigm to structure programs so that properties and behaviors are bundled into individual objects.

In [ ]:
```
x = 'A string is an object'
dir(x)
...
```

All elements in python are objects, and more specifically class objects

## Class

Think of objects as container packages with information and actions that act on the information. Most objects in python are organized in classes and contain:

1. Attributes
2. Properties
3. Methods

In [ ]:
```
class MyFirstClass:
    pass
...
```

As a convention, classes are named in a Camel Case fashion. These kind of conventions (Found in the PEP guides) helps differentiate classes from functions, etc, with a simple glance of the code.

To construct more complex classes, we will use a function within the class, called `constructor` . In Python, this is defined through the function name `__init__` .

In [ ]:
```
class MyFirstClass(object):
    def __init__(self, num1, num2):
        self.a = num1
        self.b = num2

instance = MyFirstClass(10, 100)
print(instance)
```

In [ ]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

### The correct way to instantiate the above  Dog  class is:

yes `Dog.__init__("Rufus", 3)`

no `Dog.create("Rufus", 3)`

go slower `Dog("Rufus", 3)`

go faster `Dog()`

## Attributes

Attributes are data variables within an object. In our example  a  and  b  are attributes of  aclass :

In [ ]:
```
# Dot accession
instance.a
...
```

You can access any of the attributes of an instance (the variable holding the class) by calling it after a dot. As another example:

```
i = MyFirstClass(10, 29)
i.a # access 10
i.b # access 29
```

Within the class, the attributes can be accessed through the variable self. For now, let's check on a special kind of attribute called properties.

Why bundle variables as attributes? The short answer is to keep track of variables that make sense toguether:

In [ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

persons = [Person('Chad', 28), Person('Fred', 20), Person('Claudia', 30),
           Person('Sergio', 38)]
maxage = max([i.age for i in persons])
oldest = [i.name for i in persons if i.age == maxage][0]
print("{} is the oldest person at {} years old".format(oldest, maxage))
```

You might be tempted to say that the same thing can be made with tuples (for unmutable relationships) or list (for mutable relationships). However, what if you want to mutate these variables according to one another? It will become increasingly difficult to do this with built in data structures such as lists and tuples. When attributes can be (or must be) modified on assignment, we call them properties.

## Hidden vs Exposed Attributes

In Python, we use double underscore before the attributes name to make them inaccessible/private or to hide them

In [ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.__half_life = self.age / 2

p = Person('Sergio', 38)
...
```

As you can see, in Python (differring from other languages), the hidden attributes can be accessed, but you should not, since it is supposed to be an internal attribute that would be accessed by other methods. We wil get back at this, when exploring objects.

In [ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.__half_life = self.age / 2
```

### Given the above code, what is the attribute?

yes `name`

no All

go slower `self.__half_life`

go faster `age`

## Properties

When an attribute triggers a function to set, get or delete, it is called a property:

In [ ]:
```
class MyFirstClass(object):
    def __init__(self, num1, num2):
        self.a = num1
        self.b = num2
    @property
    def z(self):
        return self.__z
    @z.setter
    def z(self, num):
        self.__z = self.a ** (self.b * num)
    @z.deleter
    def z(self):
        del self.__z

...
```

With the property, we can process an input before it populates our instance attribute through setters (the `@z.setter` in the example), getters (the `@property` in the example), and deleters (the `@z.deleter` in our example).

### Why properties?

Properties are very useful, especially when you are developing interactions with your class that require changing some of the attributes. Let's see a more applied example. Let's say we want to have a temperature converter:

In [ ]:
```
class Temperature(object):
    def __init__(self):
        self.kelvin = 0
        self.celsius = -273
        self.farenheit = 459.4
    @property
    def kelvin(self):
        return self.__kelvin
    @kelvin.setter
    def kelvin(self, kelvin):
        self.__kelvin = kelvin
        self.__celsius = self.__kelvin - 273
        self.__farenheit = (self.celsius * 1.8) + 32

...
```

As we can see, all the attributes can be update it by setting one. We could set all other attributes such as `celsius` and `farenheit` as properties, and update the rest:

```
class Temperature(object):
    def __init__(self):
        self.kelvin = 0
        self.celsius = -273
        self.farenheit = 459.4
    @property
    def kelvin(self):
        return self.__kelvin
    @kelvin.setter
    def kelvin(self, kelvin):
        self.__kelvin = kelvin
        self.__celsius = self.__kelvin - 273
        self.__farenheit = (self.celsius * 1.8) + 32
    @property
    def celsius(self):
        return self.__celsius
    @celsius.setter
    def celsius(self, celsius):
        self.__celsius = celsius
        self.__kelvin = self.__celsius + 273
        self.__farenheit = (self.__celsius * 1.8) + 32
    @property
    def farenheit(self):
        return self.__farenheit
    @farenheit.setter
    def farenheit(self, farenheit):
        self.__farenheit = farenheit
        self.__kelvin = (self.__farenheit - 32) * 0.555 + 273.15
        self.__celsius = (self.__farenheit - 32) * 0.555
```

This way, you can ask our temperature converter to update all three attribute by setting only one of them.

As you probably already noticed, the setters, getters and deleters are function that can access all attributes on the class. When the functions within the class are not setting, deleting or getting attributes, are called methods.

## Methods

TL-DR, methods are functions within a class with two major differences:

1. The method is implicitly used for an object for which it is called.
2. The method is accessible to data that is contained within the class.

The setters, getters and deleters, are actual methods applied to specific attributes.

Let's go back to our simple aclass without properties, and say you want to create methods that return values:

In [ ]:
```
class MyFirstClass(object):
    def __init__(self, num1, num2):
        self.a = num1
        self.b = num2

    def amethod(self, num3):
        return self.a * num3

    def bmethod(self, num4):
        return self.b / num4
x = MyFirstClass(1,2)
...
```

So methods are functions that belong to an object (class) and that have access to all attribute and other methods in the class. You can modify attributes through the call of methods, by setting the attribute on call.

In [ ]:
```
class Alphabet:
    def __init__(self, value):
        self._value = value
    @property
    def value(self):
        return self._value
    @value.setter
    def value(self, value):
        print('Setting value to ' + value)
        self._value = value
x = Alphabet('Peter')
x.value = 'Diesel'
```

### Given the above code, what would be printed to screen?

yes `Diesel`

no First `Setting value to Peter` then `Setting value to Diesel`

go slower `Setting value to Diesel`

go faster `Peter`

## Exploring Objects in Python

There are different ways to inspect objects (including classes) in Python:

1. `vars` : Function that returns the exposed attributes
2. `dir` : Function that returns a list of attributes (including private ones) and methods belonging to an object

In [ ]:
```
x = MyFirstClass(10, 20)
vars(x)
...
```

As shown, the `vars` function actually returns a hidden attribute called `__dict__` . When we build a class with certain set of exposed attributes, `__dict__` gets populated automatically. All Python classes will, by default have a `__dict__` attribute. However, not all **objects** have one.

By now you are probably asking, What if I want to see all methods, and attributes? You use `dir` :

In [ ]:
```
x = MyFirstClass(10, 20)
dir(x)
...
```

We can modify the behaviour of these hidden attributes and methods, by defining them within the class:

In [7]:
```
class MyFirstClass(object):
    """
    This is the first class I have created
    """
    def __init__(self, num1, num2):
        self.a = num1
        self.b = num2
    def __str__(self):
        return "This class contains {a} and {b}".format(**vars(self))
    def amethod(self, num3):
        return self.a * num3
    def bmethod(self, num4):
        return self.b / num4
x = MyFirstClass(10, 20)
str(x)
```

Out[7]:
```
'This class contains 10 and 20'
```

for more advanced inspections you can explore the `inspect` module.

### Is the following Python code correct?

In [ ]:
```
class A:
    def __init__(self,b):
        self.b=b
    def display(self):
        print(self.b)
obj=A("Hello")
del obj
```

- True
  yes
- False
  no

# Base problem

## Scenario:

You are asked to create a script that based on a temperature value in Celsius input by the user, it will return the conversion to Farenheit.

**Aim:**

Create a script to calculate Farenheit degrees based on Celsius, using the user input.

## Steps

1. In a file called `conversion.py` store the Temperature class

In [ ]:
```
class Temperature(object):
    def __init__(self):
        self.__kelvin = 0
        self.__celsius = -273
        self.__fahrenheit = 459.4
    @property
    def celsius(self):
        return celsius.__kelvin
    @celsius.setter
    def celsius(self, celsius):
        self.__celsius = float(celsius)
        self.kelvin = self.__celsius + 273
        self.fahrenheit = (self.__celsius * 1.8) + 32
```

1. Add the collection of the user's input, using the `input` function

In [ ]:
```
temp = input("Input degrees in Celsius: ")
```

1. Create the converter instance and set the input

In [ ]:
```
converter = Temperature()
converter.celsius = temp
```

1. Add printing the output of the conversion in Farenheit

```
In [ ]: fahr = converter.fahrenheit
        print("{} Celsius is {} Fahrenheit".format(temp, fahr))
```

1. Save the file and execute by:

```
python3 conversion.py
```

# Extended problem


Your assignment is to modify the script so that the user can input values in **ANY** unit (Kelvin, Celsius, and Farenheit), and the output should be all the rest. For example if the user gives you Celsius, you output the values in Farenheit and Kelvin. **Bonus:** The output should be print from the string representation of the class