Object-Oriented Programming for Data Science

University of Denver
Neba Nfonsang

January 2020

Introduction

- Python supports several programming paradigms including procedural, object-oriented, and functional programming. Python is primarily a procedural programming language, and optionally object-oriented and functional.
- Though object-oriented programming is optional in Python, understanding how it works is not optional especially if you consume or read the work of other programmers who may use OOP. In fact, OOP becomes more useful for building larger programs or software.
- You would also notice that most Python packages are built in OOP, so it is important to get comfortable writing or using code in OOP.
- The key idea about OOP is that it provides a way of organizing your code for code reuse and to minimize redundancy.
- In procedural programming, you think in terms of algorithms and what you want your program to do. In OOP, you think in terms of what objects you want your program to represent. For example, do you want the program to represent employees, cars, houses, circles?
- This lesson will focus on how to use classes to create objects or custom data structure that have attributes and methods. More advanced concepts of object-oriented programming such as

encapsulation, polymorphism, inheritance, privacy, dunder or magic methods, property decorators will be covered.

What is a Class in OOP

- A class is an abstraction or description of all the objects it defines.
- A class is a blue print, representing objects in a broader or general sense. A class is a category or type of an object.
- A class provides the definition of how the objets in that class look like (attributes) and how they behave (methods).
- A class is used to create objects just like a cookie cutter is used to create cookies.
- For example, an animal class could be used to create objects such as dogs and cats that have attributes such as names, color, etc and methods such as the sound the object makes.

What is an Object?

- Remember that everything in Python is an object.
- In OOP, there are class objects (simply called classes) and instance objects (simply called instances).

Class Objects

- A class object is created when a class statement in a class definition runs.
- When a class object object is created, it is assigned a name, which is the name of the class in the header of the class definition.
- A class object supports two operations: attribute referencing (obj.attr) and instantiation (Classname()).

- A top-level assignment inside a class statement creates a class attribute (in a more general sense, attributes are data attributes and methods).
- Class instantiation is when a class object is called or run as though it was a function (Classname()).
- Running a class object creates an instance object. Classes are a kind of factory for creating multple instances.

Instance Object

- An instance object is created when a class object runs, that is through class instantiation.
- When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()`, to initiaze the instance attributes.
- An instance object supports only one operation: attribute referencing (obj.attr: where attr is a data attribute or a method of the instance)
- For example, all strings are instances of the class str and all lists are instances of the class list. So, Python’s built-in data types such as str, int, float, list, tuple, set, etc are built-in Python classes.

Examples of Objects

- Customers, students, people, etc, can be modelled as human objects in OOP.
- Some objects could be physical such as airplanes and cars while others could be conceptual, mathematical or geometric objects such as points, lines, polygons.

Modeling Objects in OOP

- The focus of OOP is objects. OOP aims at identifying real-world objects and creating programs that model these real-world objects.

- Once the objects are identified, their basic properties (attributes) are determined.
- It is also necessary to determine what the objects can do (behaviors or methods) or what can be done to the objects.

Attributes

- There are two types of attributes in OOP: instance attribute or class attribute.
- In a more narrow sense, attribute here refers to data attributes though attributes in OOP could generally refer to both data attributes and methods in a more general sense.
- Attributes could be viewed as the properties or characteristics of a class or an object. So, attributes describe the state of a class or object.

Class Attributes

- Class attributes reference data shared by all objects created from the class.
- A class attribute is created through an top-level assignment within the class statement.
- The value of class attributes are automatically inherited by all objects of the class.
- The value of a class attribute can be accessed (or even changed) through the class object or through the instance object (obj.attr).
- If the class attribute value is changed through an instance object, it does not affect the value assigned to the class attribute.
- If the value of the class attribute is change through the class object, it does not affect instance objects that were already created but would affect the class attribute values of subsequent instance objects.
- For example, an instance attribute could be created to reference and track the number of instances for that class. This is class level information that is not specific to any instance.

Instance Attribute

- Instance attributes are variables that reference data associated to each instance.
- An instance attribute is created through an assignment to self.attr within an instance method such as the `__init__()` method.
- The value of an instance attribute is accessed through the instance object (obj.attr).
- The value of an instance attribute may vary from instance object to instance object.
- Examples of instance attributes include name, title, gender, and age of an employee object from an Employee class.

Methods

- Generally, methods are functions created inside a class.
- There are three types of methods: **instance method, class method and static method**

Instance Methods

- These are methods that are not preceeded by any decorator and thier first parameter is "self" (conventionally). The self parameter references the object iteself. Self is a required parameter when defining an instance method. The instance object is automatically passed into the self parameter when the method is called. So, the instance object does not need to be explicitly passed into the self parameter during the method call.
- There are different types of instance method, including: the ._*init*_() mutator or setter method, accessor or getter method, state representation method,other methods, and magic or dunder methods.

The ._*init*_() method

- This method is also called the constructor or initializer method. Though the ._*init*_() method is optional, it serves the purpose of initializing the instance attributes at once without the need for using several methods to initialize instance attributes. This constructor is invoked to run each time the class object runs as a function (Classname()).
- The form of the ._*init*_() method is:

```
def __init__(self, attr1, attr2, attr2,...):
  self.attr1 = attr1
  self.attr2 = attr2
  self.attr3 = attr3
  ...
```

- Some parameters of the initializer method could be take default values just like in regular functions:

```
def __init__(self, attr1, attr2=None):
  self.attr1 = attr1
  self.attr2 = attr2
  self.attr3 = []
  ...
```

- attr2 is a parameter of the initlizer with a default value and could be updated when creating the instance object.
- Though attr3 is initialized with a default value, it is not a parameter of the initializer. So, an argument cannot be provided for attr3 when creating an instance object.

Mutator or Setter Methods

- These are methods used to change, update or modify the values of instance attribute. Therefore, setter methods do not return any values.
- Setter methods take the form:

```
def set_attr(self, attr):
  self.attr = attr
```

Accessor or Getter Methods

- These are methods used to access and retrieve the values of instance attributes.Therefore, getter methods return a value.
- Getter Methods take the form:

```
def get_attr(self):
  return self.attr
```

Other Methods:

- Other methods can be used to process instance attributes to return some other new values or information about the instance object. For example, given instance attributes such as length and witdth of a room, a method can be defined to return the area of the room.

```
def monthly_payment(self):
  return self.length*self.width
'''it is assumed that the self.length and self.width attributes have been ini
tialized, for example under the initializer method.'''
```

Magic or Dunder Methods

- These are special methods that begin with a double underscore. The word "dunder" actually stands for double underscore.
- The ._*init*_() method already discussed is a dunder method. There are many other dunder methods as follows:

State Representation Methods

- This a method used to return or report the values of the instance variables.
- _*str*_() and _*repr*_() are examples of the state representation methods.
- _*str*_() and _*repr*_() methods return fairly similar information but sometimes, information displayed through _*str*_() is more human-readable.
- Given an Employee class with instance attributes such as name and position of employee, the _*str*_() representation could be defined as:

```
def __str__(self):
    return (f"name: {self.name}\n"
            f"position: {self.position}")
```

- The _*str*_() method can be called by simply printing the instance object. str(obj) or obj._*str*_() will also call the _*str*_() method.
- In an interactive interpreter, running the instance object without a print statement or running repr(obj) function calls the _*repr*_() method.

Dunder Comparison Methods

- Some dunder methods can be used to compare instance objects directly using comparison operators such as obj1 == obj2, obj1 > obj2, obj1 <= obj2, etc. The comparison is based on the atrribute value(s) of the objects. For example, circles can be said to be equal if their radius are the same. So, we can use the value of the attribute, radius, to compare circles. It is possible to use more than one attribute value.

- The following are examples of dunder methods for comparison:

```
# equality: to check whether two objects are equal
def __eq__(self, other):
  return self.attr == other.attr
```

- If comparison is based on two attribute values (same idea applies for more attribute values) use the "and" operator:

```
# equality: comparison based on several attribute values
def __eq__(self, other):
  return self.attr1 == other.attr1 and self.attr2 == other.attr2
```

- To make sure that we are comparing objects from the same class, we can use a conditional statement as follows:

```
# equality: ensure that instances compared are from the same class
def __eq__(self, other):
  if isinstance(other, self.__class__):
      return self.attr == other.attr
  else:
      return False
```

- Similarly, other comparison dunder methods could be defined as follows:

```
# not equal to
def __ne__(self, other):
  if isinstance(other, self.__class__):
      return self.attr != other.attr
  else:
      return False
```

```python
# strictly greater than
def __gt__(self, other):
    if isinstance(other, self.__class__):
        return self.attr > other.attr
    else:
        return False

# greater than or equal to
def __ge__(self, other):
    if isinstance(other, self.__class__):
        return self.attr >= other.attr
    else:
        return False

# strictly less than
def __lt__(self, other):
    if isinstance(other, self.__class__):
        return self.attr < other.attr
    else:
        return False

# less than or equal to
def __le__(self, other):
    if isinstance(other, self.__class__):
        return self.attr <= other.attr
    else:
        return False
```

Class Method

- This is a method that is preceded by the @classmethod decorator and its first parameter is cls (by convention, could be any name that is not a reserved word), which references the class itself.

```
class Employee():
  num_instances = 0
  def __init__(self, name, position):
      self.name = name
      self.position = position
      Employee.num_instance = Employee.num_instances + 1
  #class method that counts the number of instances in a class
  @classmethod # to make the method that follows a class method.
  def instance_counts(cls):
      return f"Number of instances: {Employee.num_instances}"
```

- Apart from using the @classmethod decorator to create a static method, the classmethod() special built-in function can be used.

```
class Employee():
  num_instances = 0
  def __init__(self, name, position):
      self.name = name
      self.position = position
      Employee.num_instance = Employee.num_instances + 1
  # an alternative way to create a class method
  # without using the @classmethod decorator
  def instance_counts(cls):
      return f"Number of instances: {Employee.num_instances}"
  # use the classmethod() function instead
  instance_count= classmethod(instance_counts)
```

- A static method can be called through the class object or through the instance object: obj.methodname()

Static Method

- This is a method that is preceeded by the @staticmethod decorator, and it's first parameter does not reference the class object or instance object.
- Static class methods can be used to manage or process class data or class attribute values.
- Apart from using the @staticmethod decorator to create a static method, the staticmethod() special built-in function can be used.

```python
class Employee():
    num_instances = 0
    def __init__(self, name, position):
        self.name = name
        self.position = position
        Employee.num_instance = Employee.num_instances + 1
    #static method that counts the number of instances in a class
    @staticmethod # to make the following method a static method
    def instance_counts():
        return f"Number of instances: {Employee.num_instances}"
```

- A static method can be called through the class object or through the instance object: obj.methodname()

```python
class Employee():
    num_instances = 0
    def __init__(self, name, position):
        self.name = name
        self.position = position
        Employee.num_instance = Employee.num_instances + 1
    # an alternative way to create a static method
    # without using the @staticmethod decorator
    def instance_counts(): # does not take cls or self
        return f"Number of instances: {Employee.num_instances}"
    # use the staticmethod() function instead
    instance_count= staticmethod(instance_counts)
```

OOP in Action

A Simple Class and an Instance

In [1]:

```python
# create a simple class and an instance
class Employee():
    pass

# create an empty instance object using the class
e1 = Employee()

# add some data to the instance
e1.name = "John"
e1.position = "Python Developer"

# access the values of the attribute directly
print(e1.name)
print(e1.position)
```

```
John
Python Developer
```

In [2]:

```python
# check the class of the instance
print(e1.__class__)
```

```
<class '__main__.Employee'>
```

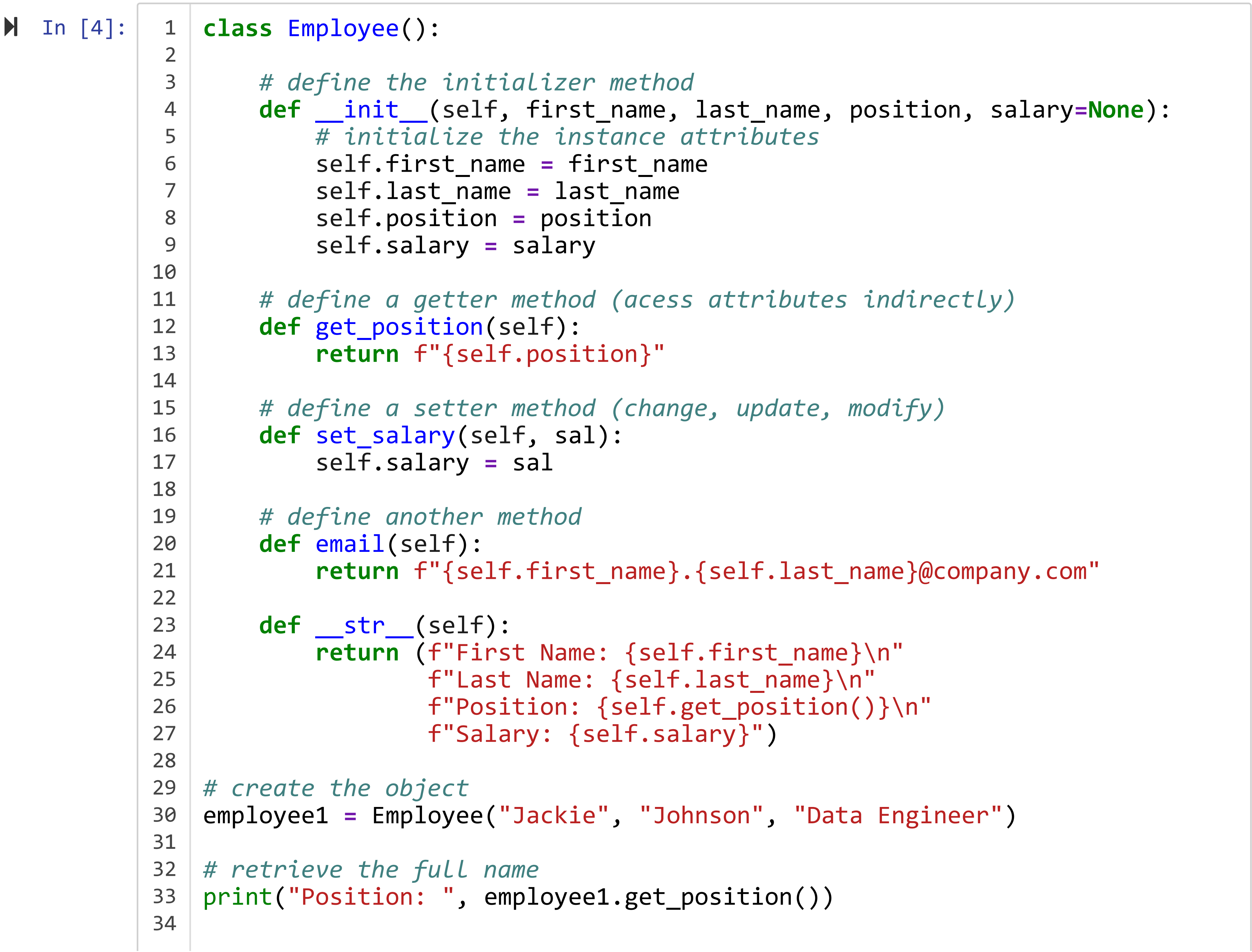
In [3]:

```python
# check if the instance e1 belongs to Employee class
isinstance(e1, Employee)
```

Out[3]:

```
True
```

A Class with Different Instance Methods



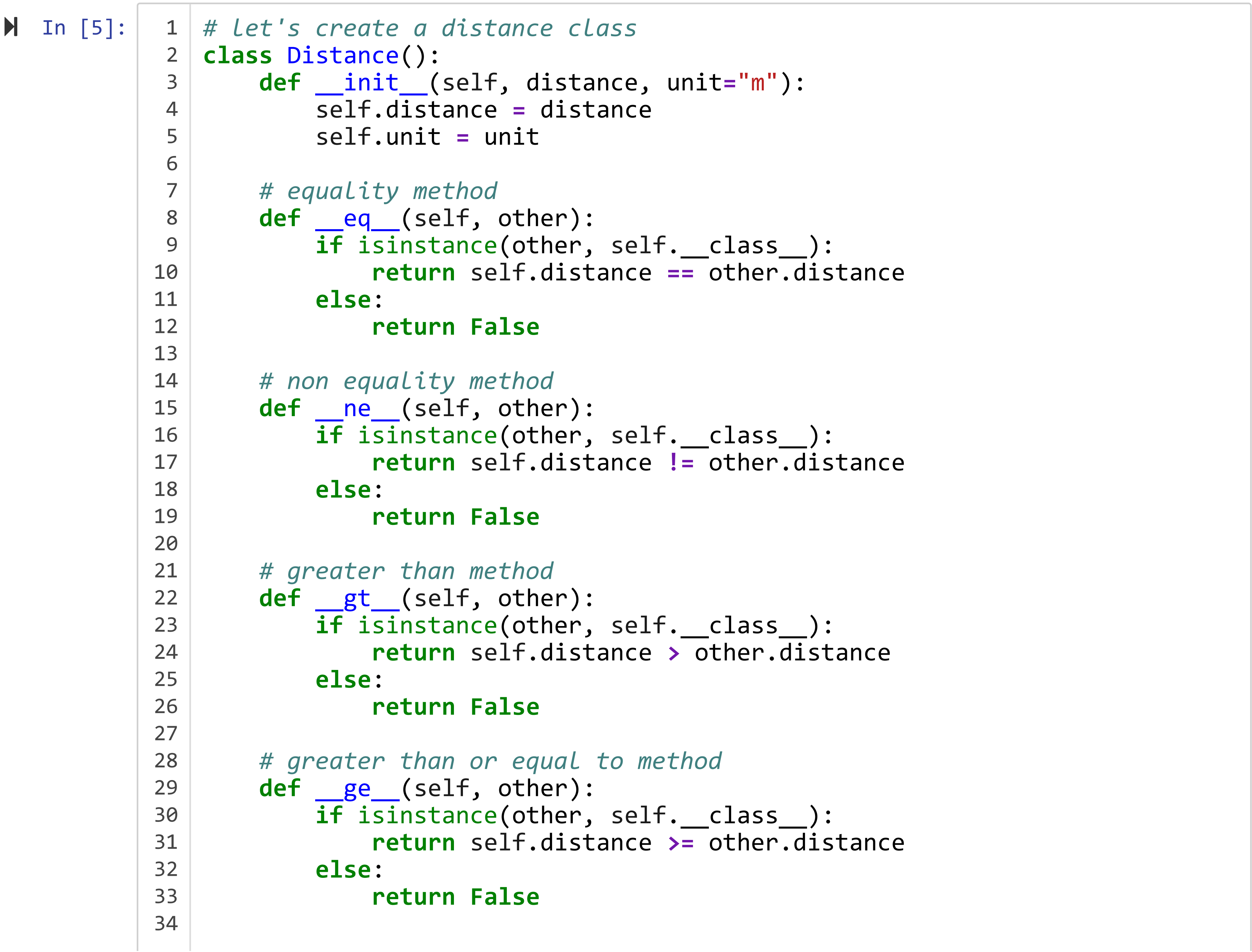
In [4]:

```python
class Employee():

    # define the initializer method
    def __init__(self, first_name, last_name, position, salary=None):
        # initialize the instance attributes
        self.first_name = first_name
        self.last_name = last_name
        self.position = position
        self.salary = salary

    # define a getter method (acess attributes indirectly)
    def get_position(self):
        return f"{self.position}"

    # define a setter method (change, update, modify)
    def set_salary(self, sal):
        self.salary = sal

    # define another method
    def email(self):
        return f"{self.first_name}.{self.last_name}@company.com"

    def __str__(self):
        return (f"First Name: {self.first_name}\n"
                f"Last Name: {self.last_name}\n"
                f"Position: {self.get_position()}\n"
                f"Salary: {self.salary}")

# create the object
employee1 = Employee("Jackie", "Johnson", "Data Engineer")

# retrieve the full name
print("Position: ", employee1.get_position())

```

```python
# set salary to 98000
employee1.set_salary(98000)

# retrieve salary directly using instance attribute
print("Salary: ", employee1.salary)
print("----"*10)

# print the string representation of the instance
print(employee1)
```

```
Position:  Data Engineer
Salary:  98000
----------------------------------------
First Name: Jackie
Last Name: Johnson
Position: Data Engineer
Salary: 98000
```

A Class with Comparison Dunder Methods



```python
# let's create a distance class
class Distance():
    def __init__(self, distance, unit="m"):
        self.distance = distance
        self.unit = unit

    # equality method
    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.distance == other.distance
        else:
            return False

    # non equality method
    def __ne__(self, other):
        if isinstance(other, self.__class__):
            return self.distance != other.distance
        else:
            return False

    # greater than method
    def __gt__(self, other):
        if isinstance(other, self.__class__):
            return self.distance > other.distance
        else:
            return False

    # greater than or equal to method
    def __ge__(self, other):
        if isinstance(other, self.__class__):
            return self.distance >= other.distance
        else:
            return False

    # less than method
    def __lt__(self, other):
        if isinstance(other, self.__class__):
            return self.distance < other.distance
        else:
            return False

    # less than or equal to method
    def __le__(self, other):
        if isinstance(other, self.__class__):
            return self.distance == other.distance
        else:
            return False

x = Distance(10)
y = Distance(30)
z = Distance(10)

print("Is x equal to y? ", x==y)
print("Is x equal to z? ", x==z)
print("Is x less than to y? ", x<y)
```

```
Is x equal to y?  False
Is x equal to z?  True
Is x less than to y?  True
```

```python
# If we add x + y, we get an error message:

x + y
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-106-cd60f97aa77f> in <module>()
```

```
----> 1 x + y

TypeError: unsupported operand type(s) for +: 'Distance' and 'Distance'
```

- To to able to add values using arithmetic operators, we need to define the arithmetic dunder method inside the class.

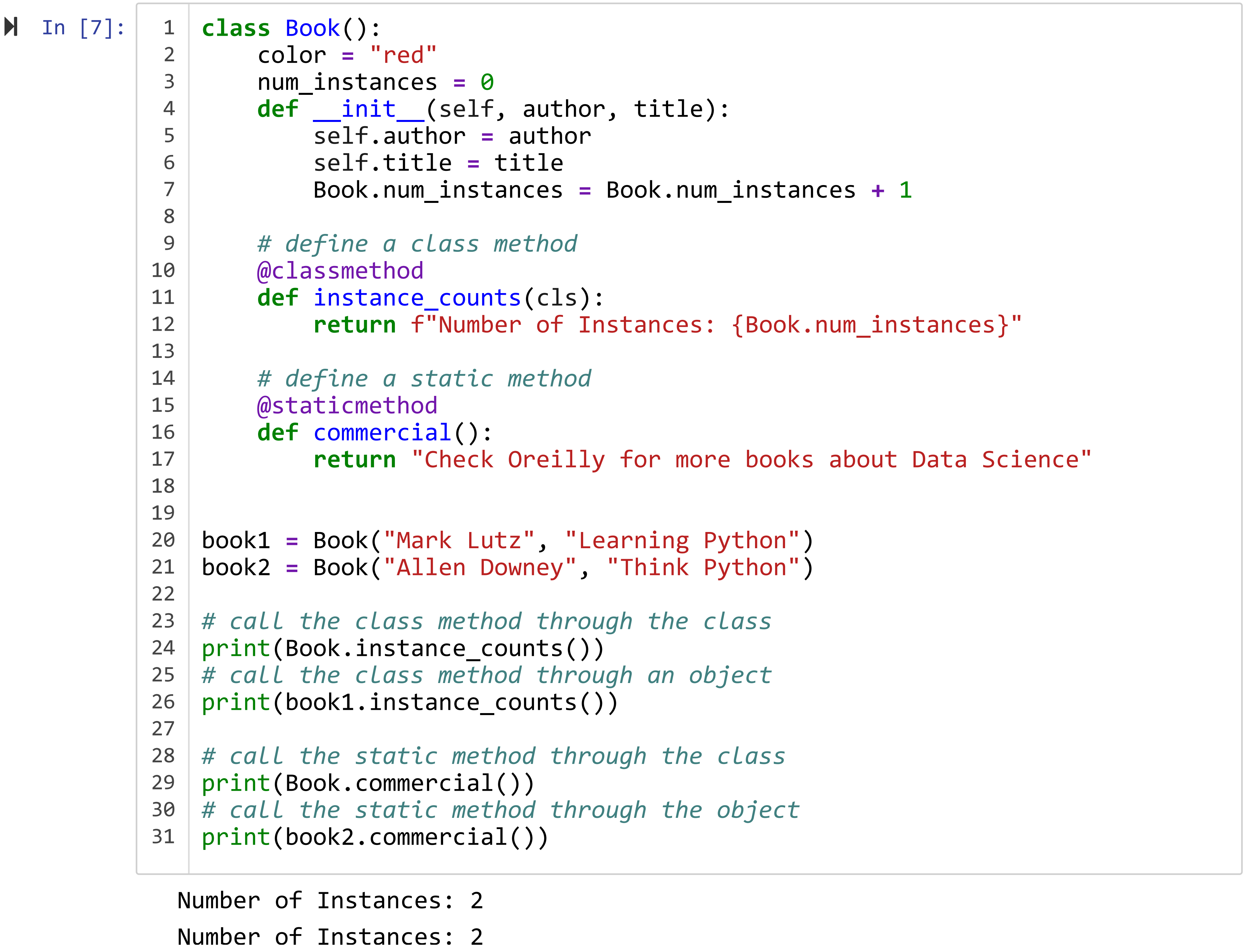
A Class with Arithmetic Dunder Methods

In [6]:

```python
# let's create a distance class
class Length():
    def __init__(self, length, unit="m"):
        self.length = length
        self.unit = unit

    # define a method that adds
    def __add__(self, other):
        return self.length + other.length

    # define a method that subtracts
    def __sub__(self, other):
        return self.length - other.length

a = Length(5)
b = Length(20)
print("a + b = ", a + b)
print("a - b = ", a - b)
```

```
a + b =  25
a - b =  -15
```

A Class with Static and Class Methods



In [7]:

```
class Book():
    color = "red"
    num_instances = 0
    def __init__(self, author, title):
        self.author = author
        self.title = title
        Book.num_instances = Book.num_instances + 1

    # define a class method
    @classmethod
    def instance_counts(cls):
        return f"Number of Instances: {Book.num_instances}"

    # define a static method
    @staticmethod
    def commercial():
        return "Check Oreilly for more books about Data Science"


book1 = Book("Mark Lutz", "Learning Python")
book2 = Book("Allen Downey", "Think Python")

# call the class method through the class
print(Book.instance_counts())
# call the class method through an object
print(book1.instance_counts())

# call the static method through the class
print(Book.commercial())
# call the static method through the object
print(book2.commercial())
```

```
Number of Instances: 2
Number of Instances: 2
```

```
Check Oreilly for more books about Data Science
Check Oreilly for more books about Data Science
```

Private Attributes in Python

- Some object oriented languages support private instance attributes that cannot be accessed directly from ouside the class.
- Python does not have private attributes so the values of instance attributes can easily be accessed using the obj.attr syntax.
- However, a bit of privacy can be acheived in Python by using getters to write values and setters to read values of attributes that the programmer wants to be private. This is not a perfect solution to making attributes private, however this provides an indirect way of accessing attributes.
- A better way to make attributes private is to use property.
- Python also hides attributes from being accessed outside the class by using double underscore at beginning of each attribute, for example, self.__name.

In [8]:

```python
# accessing attributes directly
class Person():
    def __init__(self, name):
        self.name = name
p1 = Person("John")

# access name directly
print("Name: ", p1.name)

# change name
p1.name = "Jack"

# check the name again
print("Name: ", p1.name)
```

```
Name:   John
Name:   Jack
```

In [9]:

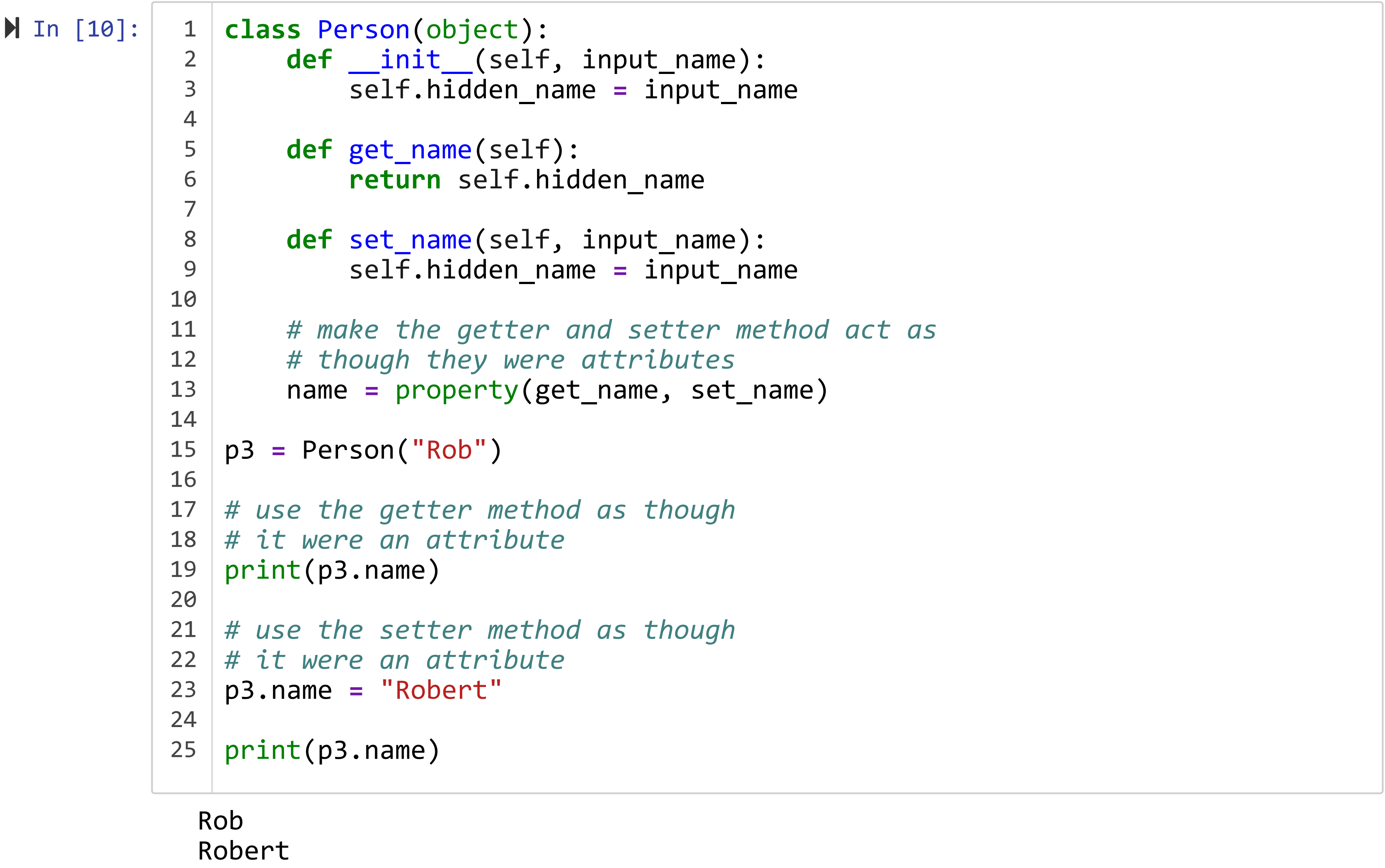
```python
# using getters and setters
# to provide a bit of privacy
class Person():
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        return self.hidden_name

    def set_name(self, input_name):
        self.hidden_name = input_name

p2 = Person("James")
print("Name: ", p2.get_name())
p2.set_name("Jacob")
print("Name: ", p2.get_name())
```

```
Name:   James
Name:   Jacob
```

- Note that obj.hidden_name will still work

Property

- Property decorator or function allow us to call getter and setter methods as though they were attributes or properties, without using the parenthesis at the end.
- Using property also allow us to acheive a certain level of privacy.
- There are two ways to achieve this, or to create a property object: by using the property decorator @property before the setter or getter method (or any method that computes a value), or by using the property() function.



```python
class Person(object):
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        return self.hidden_name

    def set_name(self, input_name):
        self.hidden_name = input_name

    # make the getter and setter method act as
    # though they were attributes
    name = property(get_name, set_name)

p3 = Person("Rob")

# use the getter method as though
# it were an attribute
print(p3.name)

# use the setter method as though
# it were an attribute
p3.name = "Robert"

print(p3.name)
```

```
Rob
Robert
```

- Note that the same property *name* has replaced the getter and setter methods
- Also note that obj.hidden_name will still work

In [11]:

```python
class Person(object):
    def __init__(self, input_name):
        self.hidden_name = input_name
    @property
    def name(self):
        return self.hidden_name
    @name.setter
    def name(self, input_name):
        self.hidden_name = input_name

p4 = Person("Suzzie")
# use the getter method as though
# it were an attribute
print(p4.name)

# use the setter method as though
# it were an attribute
p4.name = "Suzzane"

print(p4.name)
# note that the setter and getter methods
# are defined with the same name.
```

```
Suzzie
Suzzane
```

- The obj.hidden_name still works

Hidding Attributes Using Double Underscores

In [12]:

```
class Person(object):
    def __init__(self, input_name):
        self.__name = input_name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, input_name):
        self.__name = input_name

p5 = Person("Mary")
print("Name:  ", p5.name)
p5.name = "Maria"
print("Name:  ",p5.name)
```

```
Name:   Mary
Name:   Maria
```

This looks good as obj._*name does not work. However, this does not completely make the attribute .*_name private. The attribute .__name can be used to access the attribute value as follows

In [13]:

```
# accessing a double underscore attribute
# through the class
p5._Person__name
```

Out[13]:

```
'Maria'
```

However, the use of double underscore achieves a certain level of privacy. In Python, when a double underscore is used at the beginnig of an attribute name, it is a way of telling other programmers that the attribute should not be accessed directly or changed. This also prevents the attributes from being changed accidentally.

Inheritance

- Inheritance is the ability of a child class to inherit the attributes and methods of a parent class. Classes support inheritance, but modules don't.
- The parent class is also called the superclass or base class
- The child class is also called the subclass or derived class.
- Inheritance allows us to extend a class by redefining it's attributes outside the class itself.
- Inheritance allows us to customize existing software by adding new methods instead of building from scratch.
- Additional attributes can be initalized for the child class through a subclass initializer.
- The initializer method initializes the instance attributes of the child class and overrides the initializer method of the parent class.
- If we want the child class to have only the attributes of the parent class, we don't need any initializer for the child class.
- The super() built-in function in Python allows us to extend the functionality of the parent class (superclass) to the child class (subclass).

In [14]:

```python
# let create the employee class again
class Employee():

    def __init__(self, first_name, last_name, position, salary=None):
        self.first_name = first_name
        self.last_name = last_name
        self.position = position
        self.salary = salary

    def get_position(self):
        return f"{self.position}"

    def set_salary(self, sal):
        self.salary = sal

    def email(self):
        return f"{self.first_name}.{self.last_name}@company.com"

    def __str__(self):
        return (f"First Name: {self.first_name}\n"
                f"Last Name: {self.last_name}\n"
                f"Position: {self.get_position()}\n"
                f"Salary: {self.salary}")

# Create a Manager Class to inherit from the Employee class
# pass Employee (parent) class into the Manager class
class Manager(Employee):
    pass
# the Manager class inherits all the functionalities of the Employee class

manager1 = Manager("Grace", "Mckinzie", "Senior Manager", 15000)
print(manager1)
print(manager1.email())
```

```
First Name: Grace
Last Name: Mckinzie
Position: Senior Manager
Salary: 15000
Grace.Mckinzie@company.com
```

- A method of the parent class is overridden by the method of the child class if the method of the child class has the same signature as the method of the parent class.
- Same signature implies same method name, same parameter names and number of parameters.

In [15]:

```python
# override a method of the parent class
# you can override any method in the parent class
# including the initializer method
class Director(Employee):

    # override the email method
    def email(self):
        return f"{self.first_name}.{self.last_name}@companydirector.com"
director1 = Manager("Nat", "Peterson", "Assistant Director", 100000)
print(director1)
print(director1.email())
```

```
First Name: Nat
Last Name: Peterson
Position: Assistant Director
Salary: 100000
Nat.Peterson@company.com
```

In [16]:

```python
# add a method to the child class
class Ceo(Employee):

    # add the full name method to the child class
    def full_name(self):
        return f"Full Name: {self.first_name} {self.last_name}"

ceo1 = Ceo("Dan", "Anderson", "Cheif Financial Executive Officer", 170000)

print(ceo1)
print(ceo1.full_name())
```

```
First Name: Dan
Last Name: Anderson
Position: Cheif Financial Executive Officer
Salary: 170000
Full Name: Dan Anderson
```

In [17]:

```
# let create another employee class
# a base salary attribute is initialized
# but it is not a parameter in the initializer in the parent class
class Employee():

    def __init__(self, first_name, last_name, position):
        self.first_name = first_name
        self.last_name = last_name
        self.position = position
        self.base_salary = 50000


    def get_position(self):
        return f"{self.position}"

    def set_salary(self, sal):
        self.salary = sal

    def email(self):
        return f"{self.first_name}.{self.last_name}@company.com"

    def __str__(self):
        return (f"First Name: {self.first_name}\n"
                f"Last Name: {self.last_name}\n"
                f"Position: {self.get_position()}\n"
                f"Base Salary: {self.base_salary}")

# create a Ceo class that inherits from the employee class
class Ceo(Employee):
    # override the initializer of the parent class
    def __init__(self, first_name, last_name, position):
        self.first_name = first_name
        self.last_name = last_name
        self.position = position
        # we did not initialize the base_salary

```

When the instance object, ceo1, is printed, we get an error messsage indicating that there is no attribute as base_salary in the ceo class. Though the Ceo class inherited the Employee class, which has the base_salary attribute, the initializer of the child or Ceo class overrode the initializer of the parent or Employee class because they have the same signature.

```
ceo1 = Ceo("Jess", "Tom", "staff")
print(ceo1)
`
```

Printing the instance object, ceo1 generates the following error:

```
---------------------------------------------------------------
AttributeError                Traceback (most recent call last)
<ipython-input-246-53fab6befddf> in <module>()
     37
     38 ceo1 = Ceo("Jess", "Tom", "staff")
---> 39 print(ceo1)

<ipython-input-246-53fab6befddf> in __str__(self)
     21
     22     def __str__(self):
---> 23         return (f"First Name: {self.first_name}\n"
     24                 f"Last Name: {self.last_name}\n"
     25                 f"Position: {self.get_position()}\n"

AttributeError: 'Ceo' object has no attribute 'base_salary'
```

- In order to invoke parent class initializer which has been overridden by the child class initializer, we need to run super().\init() under the child class initializer.
- So, generally, if a method of the child class overrode a method of the parent class and we still want to invoke the method of the parent class inside the child class, then we need to use the super() function: super.method(arg)

In [18]:

```python
# create a Ceo class that inherits from the employee class
class Ceo(Employee):
    # override the initializer of the parent class
    def __init__(self, first_name, last_name, position):
        # invoke super() to initialize all these parameters
        # from the parent class including the extra base salary attribute
        super().__init__(first_name, last_name, position)

ceo1 = Ceo("Jess", "Tom", "staff")
print(ceo1)

```

```
First Name: Jess
Last Name: Tom
Position: staff
Base Salary: 50000
```

In [19]:

```python
# any additional instance attribute in the
# child class should be intialized when super is used
# create a Ceo class that inherits from the employee class
class Ceo(Employee):
    # override the initializer of the parent class
    def __init__(self, first_name, last_name, position, service_years=5):
        # initialize the extra instance attribute
        self.service_years = service_years
        # run super() to invoke the parent class initializer
        # this will initialize the other
        # parameters also in the child intializer
        super().__init__(first_name, last_name, position)

    def set_service_years(self, yrs):
        self.service_years = yrs

    def get_service_years(self):
        return self.service_years

ceo1 = Ceo("Jess", "Tom", "staff")
print(ceo1)
```

```
First Name: Jess
Last Name: Tom
Position: staff
Base Salary: 50000
```

Polymorphism

Polymorphis in programming is when methods with the same name do different things when called objects of different types. For example, two subclasses inheriting from the same parent class can override a method of the parent class so that each method in the subclass does something different from each other.

Encapsulation

This simply means packaging in Python, or hidding implementation details. This prevents users from changing the code, but encapsulation does not necessarily imply privacy as encapsulation is more about packaging than restricting. Classes promote encapsulation, which reduces redundancy.

In [ ]:

```

```