Python Programming

Module 2: Advanced Python Programming

Lesson 1: Object-Oriented Programming

Neba Nfonsang
University of Denver

Contents

Lesson 1: Object-Oriented Programming

- Object-Oriented Programming Overview
- Classes and Objects
- Instantiation
- Attributes
- Methods

Object-Oriented Programming (OOP)

- Object Oriented Programming is a programming paradigm, approach or style that allows the use of classes to create objects that have attributes and methods.
- OOP is a one of the most efficient way of writing programs or software.
- OOP is very useful in writing large programs, however, using OOP is optional in Python.
- We have learned that Python has built-in object types.
- Programmers can use Python’s built-in types to create their own classes (types) of objects.

OOP vs. Procedural Programming

- Though OOP is optional in Python, it is still very important especially if you are writing a large or complex program.
- The standard programming approach in Python is **procedural programming.**
- In procedural programming, procedures or algorithms are used to accomplish a task.
- In procedural programming, you first think of what you want your program to do.
- In OOP, you rather first think of what you want your program to represent.

OOP vs. Procedural Programming

- In procedural programming, functions are mostly used to organize code.
- In OOP, classes are used to organize code though class still contain special functions and could have methods, which are functions belonging to objects.
- Procedural programming is okay for simple programs but as a program becomes more complex and larger, the traditional procedural programming approach becomes limited.
- OOP is more valuable for larger programs.

What are Classes in OOP

- A class is a blue print representing objects in a broader or general sense. A class is a category of an object.
- A class is an abstraction or a description of all the objects it defines.
- A class provides the definition of how the objects in that class look like (attributes) and how they behave (behaviors).
- In OOP, **Animal** could be a class that defines the attributes and behaviors of animals such as dogs and cats.

What is an object?

- In OOP, an object is an instance of a class, having attributes (properties) and methods (behaviors).
- For example, all strings are instances of the class ***str*** and all lists are instances of the class ***list***
- Note that data types are classes. Therefore, Python’s built-in data types such as ***str, int, float, list, tuple, dictionaries*** and ***sets*** are built-in Python classes.
- An object (instance object) is created when a class is called.

Examples of Objects

- Cars and airplanes are physical objects that can be modeled in OOP.
- Employees, customers, students, etc are human object that can also be modeled in OOP.
- Sometimes, conceptual or mathematical or geometric objects such as points, lines, polygons can be modeled with OOP.

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.
- Generally, real-world objects could be physical or conceptual.
- Once the objects are identified, their basic properties (attributes) are determined.
- It is also necessary to determine what the objects can do (behavior) or what can be done to the objects.

Attributes

- Attributes describe the state of the object.
- Attributes are sometimes viewed as the properties or characteristics of a class of objects.
- For example: name, title, gender, and age are the attributes of an employee.
- Attributes are generally variables that reference information or data about an object.
- There are basically two types of attributes – class attributes and instance attributes.

Attributes

- Instance attributes (also called **instance variables**) are variables that belongs to specific objects.
- The values of the data referenced by instance attributes are not necessarily the same for every object.
- For example, self.age implies that age is an instance variable belonging to an object. Different objects could have different age values.
- **Class attributes** are variables belonging to a class. The value of the class attribute is shared by all objects of that class.

Methods

- Basically, methods are functions created inside a class.
- Generally, methods are functions belonging to an object of a class.

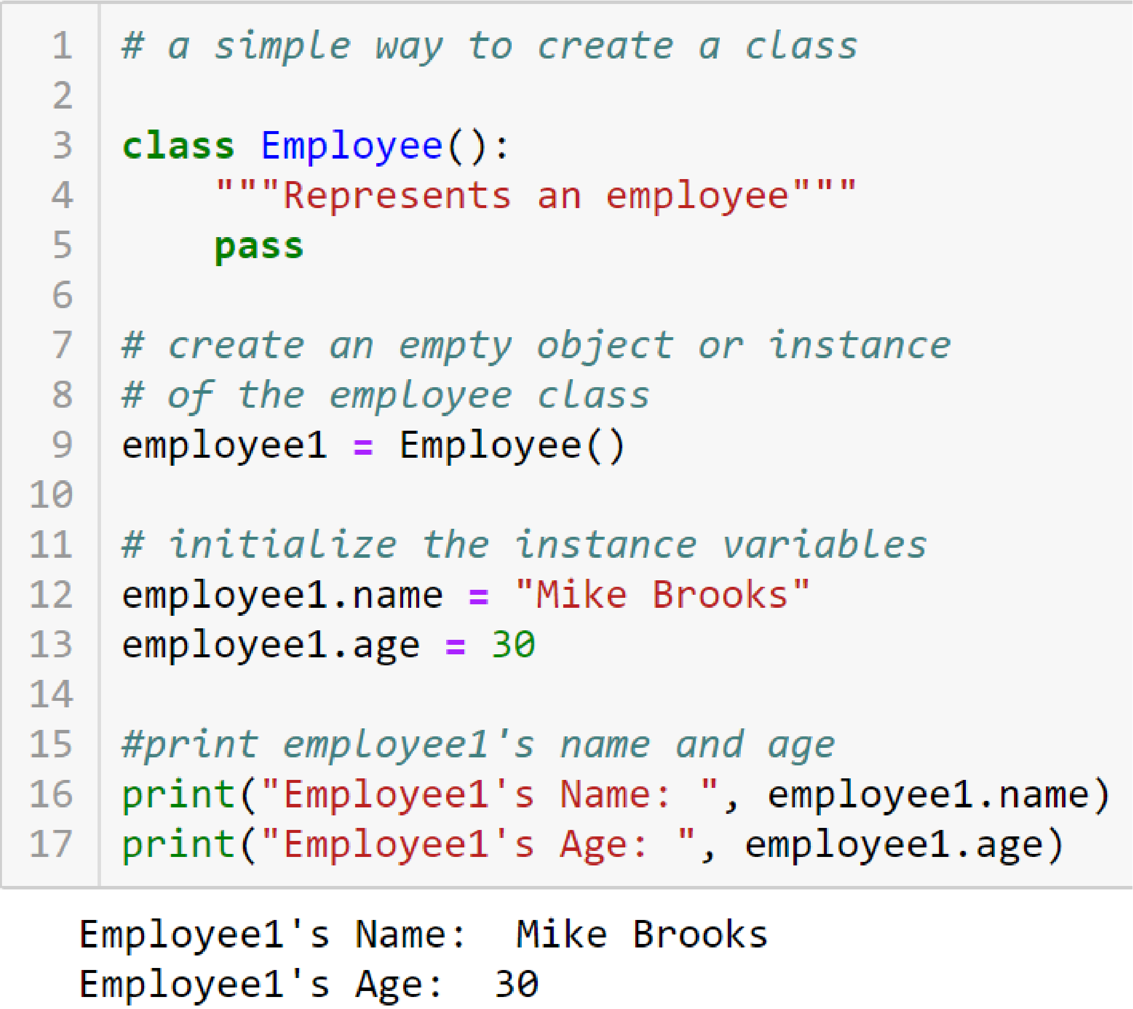
- Different kinds of methods usually used in classes are:
  - □ The __init__() method
  - □ Mutator or setter method
  - □ Accessor or getter method
  - □ State representation method
  - □ Other methods

User-Defined Classes

- Python gives us the room to create our own custom classes or data types.
- Again, each class will have methods and attributes
- A class is used to create objects just like a cookie cutter is used to create cookies.
- A class begin with a class header, which is the first line of code.
- The class header starts with the reserved word, **class**, followed by the name of the class, parenthesis and ends with a colon(:).

User-Defined Class

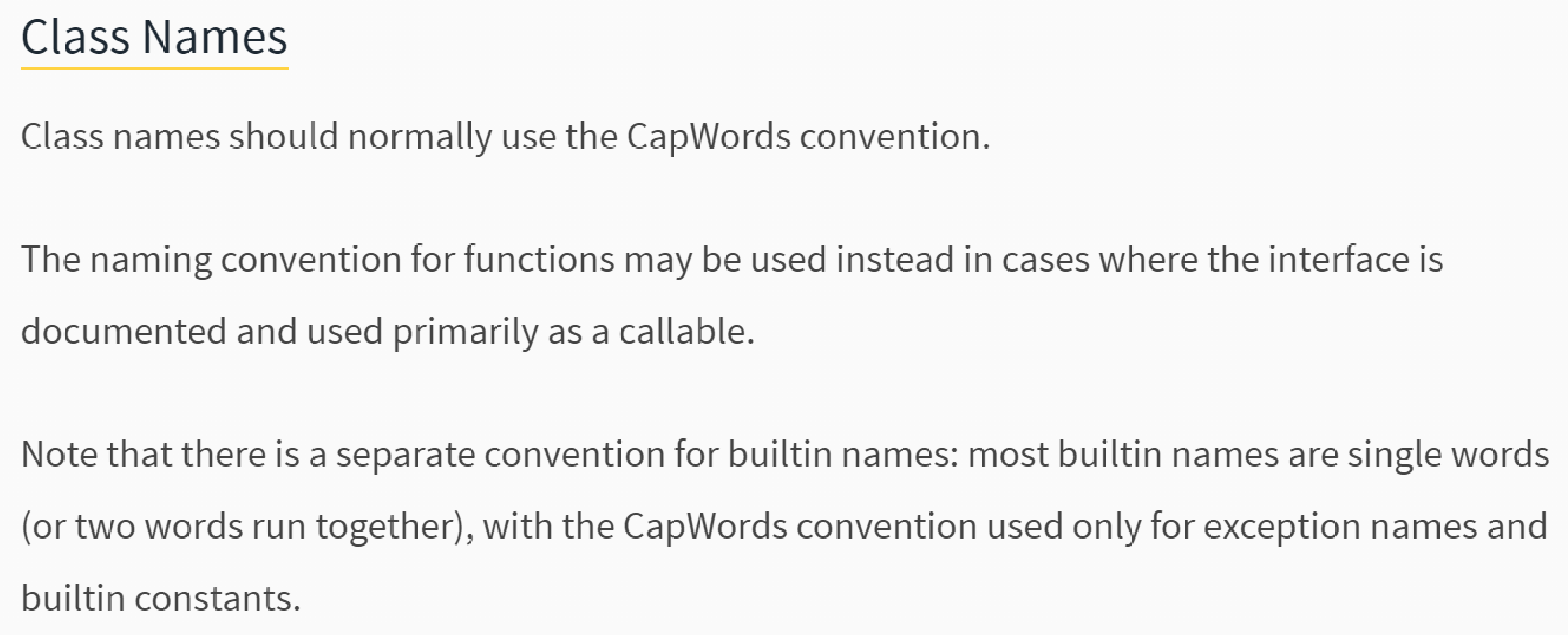
- Here is a simple way to define a class. This is not recommended but this is a good way of getting started.
- Note that CamelCase style is normally used to name classes. For example, MountainBike or Bike class



```python
# a simple way to create a class

class Employee():
    """Represents an employee"""
    pass

# create an empty object or instance
# of the employee class
employee1 = Employee()

# initialize the instance variables
employee1.name = "Mike Brooks"
employee1.age = 30

#print employee1's name and age
print("Employee1's Name: ", employee1.name)
print("Employee1's Age: ", employee1.age)
```

```
Employee1's Name:  Mike Brooks
Employee1's Age:  30
```

Good Practice for Naming Classes



Class Names

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

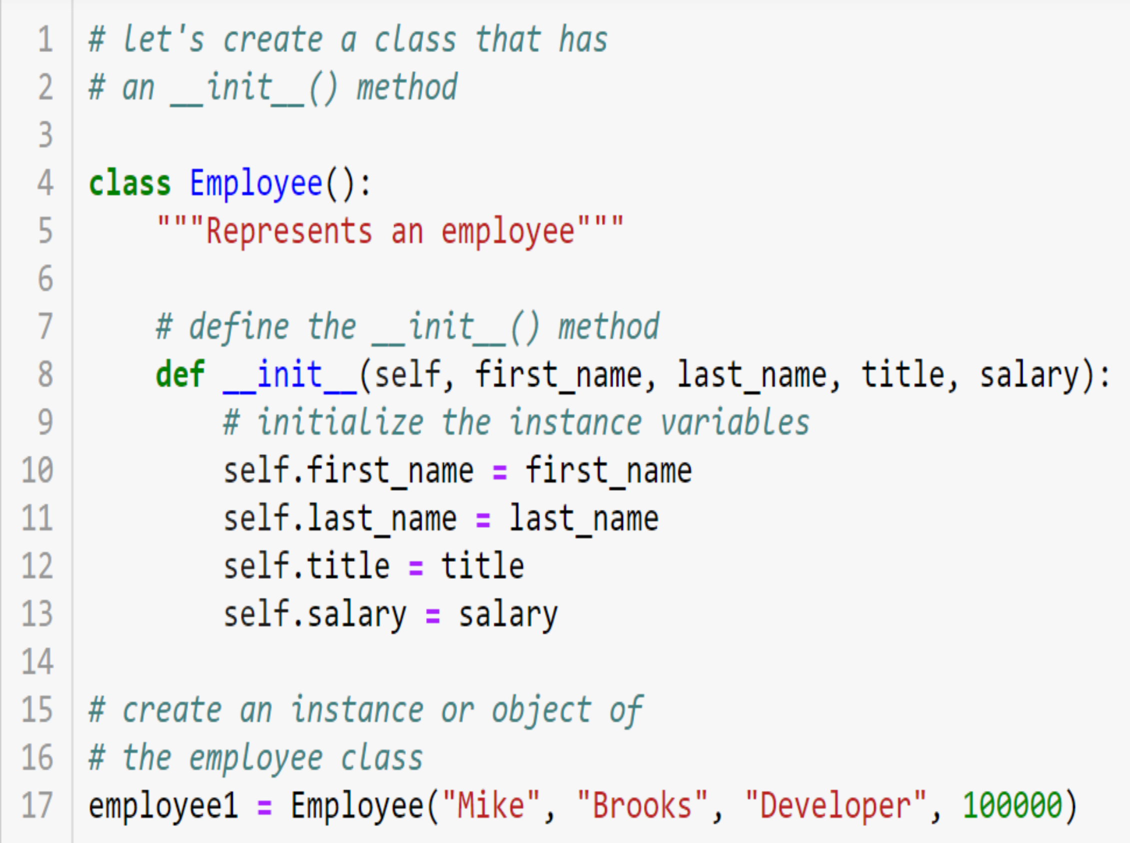
Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

PEP 8 -- Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/

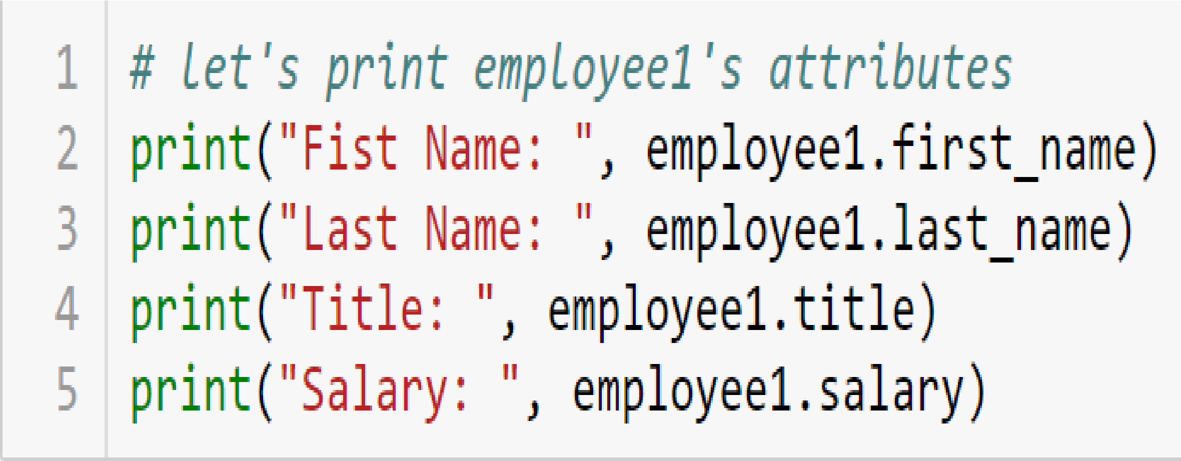
The __init__() Method

- A better way to define a class is to use the __init__() method.
- The __init__() method is a special method in a class that serves as a constructor of an object.
- It is therefore also called a constructor or the initializer method.
- The __init__() method has parameters and it’s first parameter is traditionally named **self**. The parameter self is required and reference the object itself.
- The other parameters are the instance variables referencing the data that describe the state of the object.

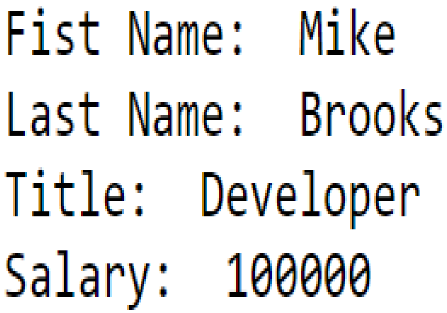
Let’s Define an __init__() Method within a Class



```python
# let's create a class that has
# an __init__() method

class Employee():
    """Represents an employee"""

    # define the __init__() method
    def __init__(self, first_name, last_name, title, salary):
        # initialize the instance variables
        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks", "Developer", 100000)
```



```python
# let's print employee1's attributes
print("Fist Name: ", employee1.first_name)
print("Last Name: ", employee1.last_name)
print("Title: ", employee1.title)
print("Salary: ", employee1.salary)
```



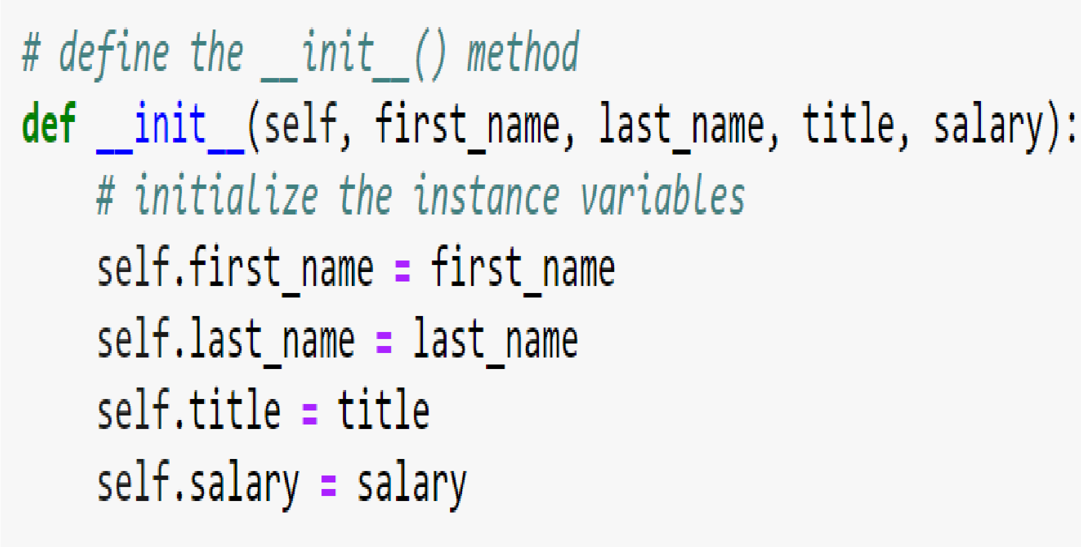
```
Fist Name:  Mike
Last Name:  Brooks
Title:  Developer
Salary:  100000
```

Instantiation

- Instantiation is the creation of an instance of a class when the class is called.
- If an _init_() method is defined in a class, each time the class is run, the _init_() method is automatically invoked.
- That means, __init__() is called during every instantiation.
- The __init__() method then creates the instance variables by assigning the values of the attributes.
- If no __init__() method is present in the class, the class call returns an empty instance, without initializing it.

Forms of the __init__() Method

- There is flexibility on how parameters of the __init__() method can be specified.
- All the attributes can be used as parameters without default values.
- Just like parameter of functions, parameters specified this way require that all the arguments be provided in the class call.

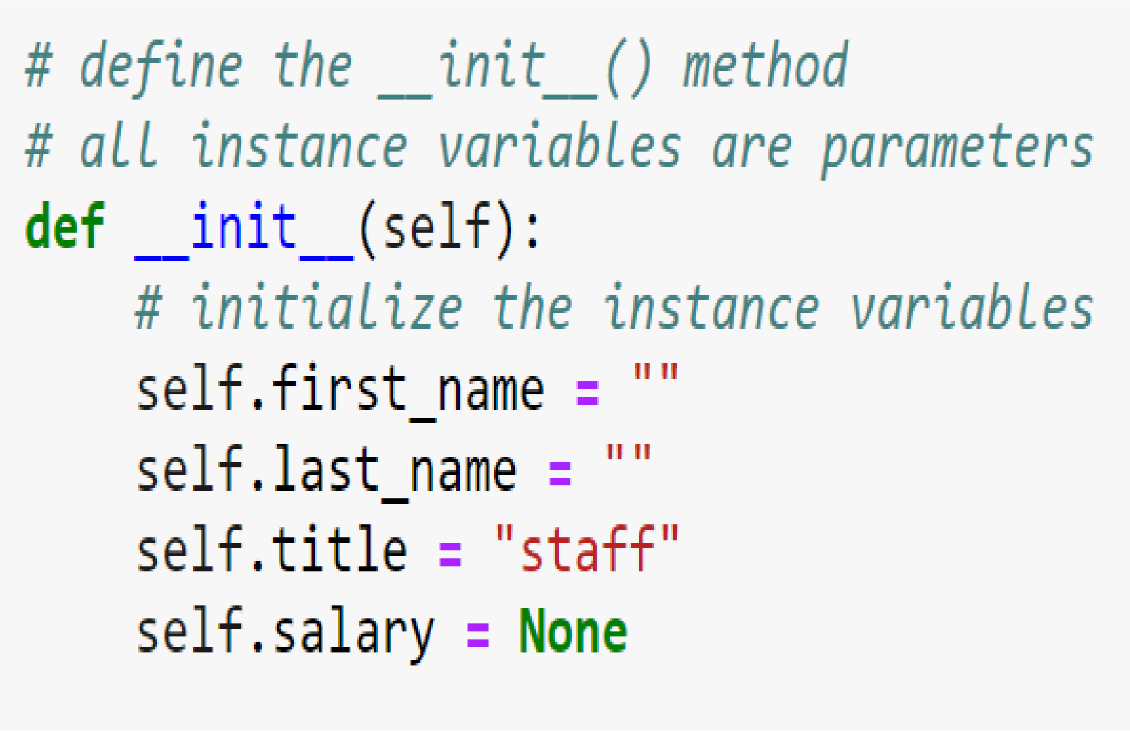


```python
# define the __init__() method
def __init__(self, first_name, last_name, title, salary):
    # initialize the instance variables
    self.first_name = first_name
    self.last_name = last_name
    self.title = title
    self.salary = salary
```

Note that after the header of the _init_ methods, the instance variables are initialized in the next indented block(s) of code.

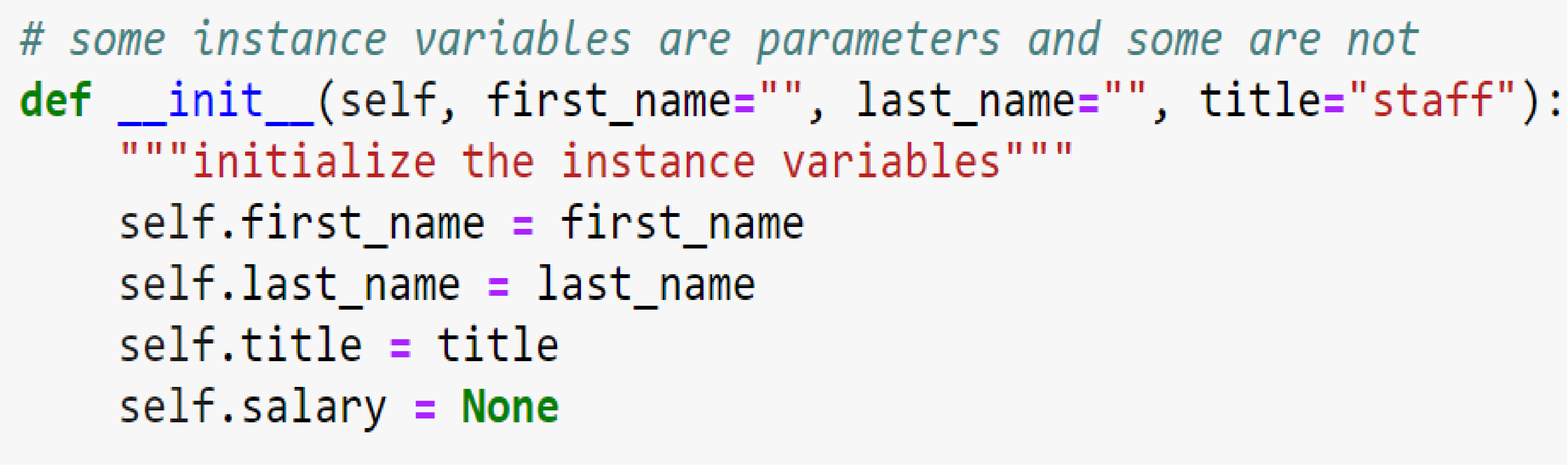
Forms of the __init__() Method

- Here is a situation where there are no parameters in the __init__() method except self.
- This form of __init__() method usually requires that a **setter** method be defined to set the values of the instance variables.



```
# define the __init__() method
# all instance variables are parameters
def __init__(self):
    # initialize the instance variables
    self.first_name = ""
    self.last_name = ""
    self.title = "staff"
    self.salary = None
```

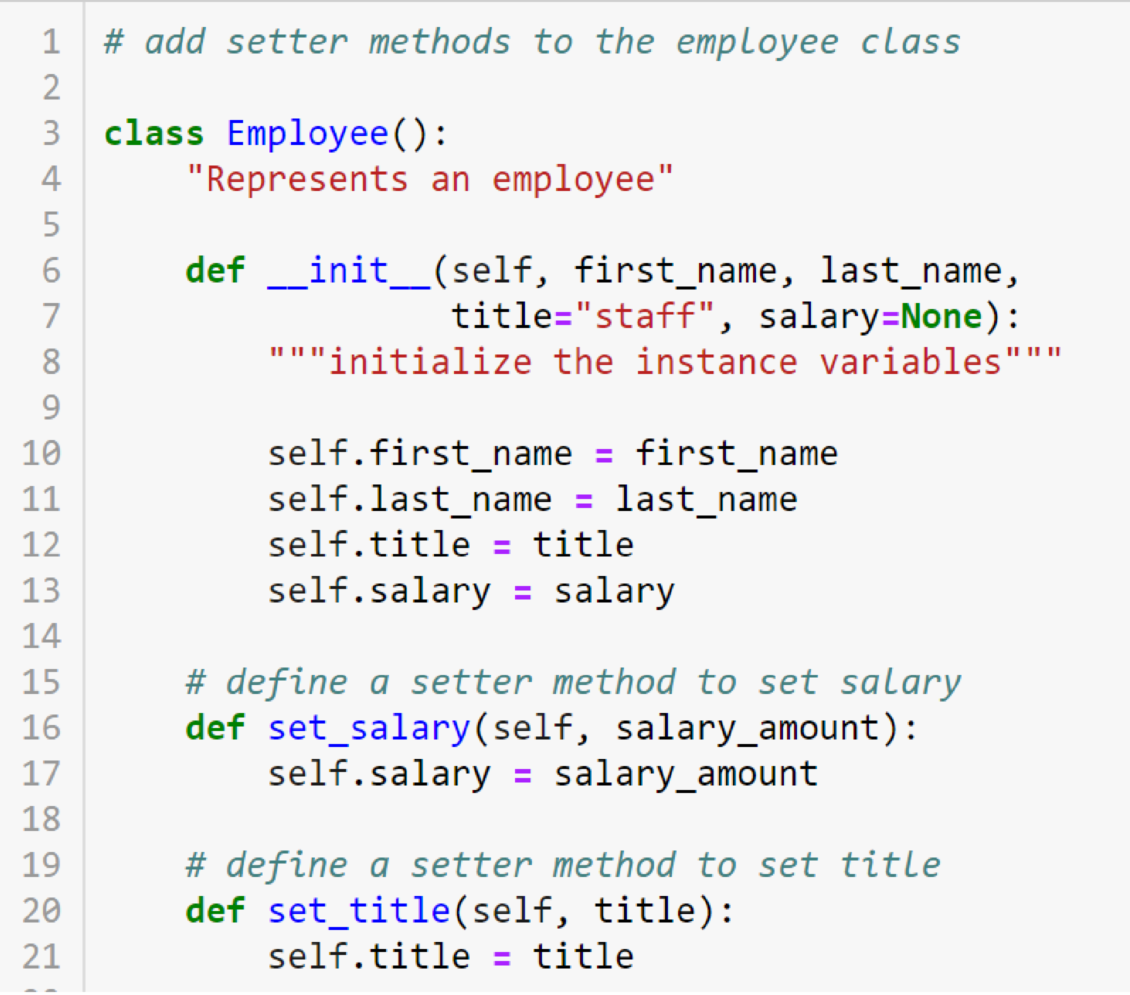
Forms of the __init__() Method



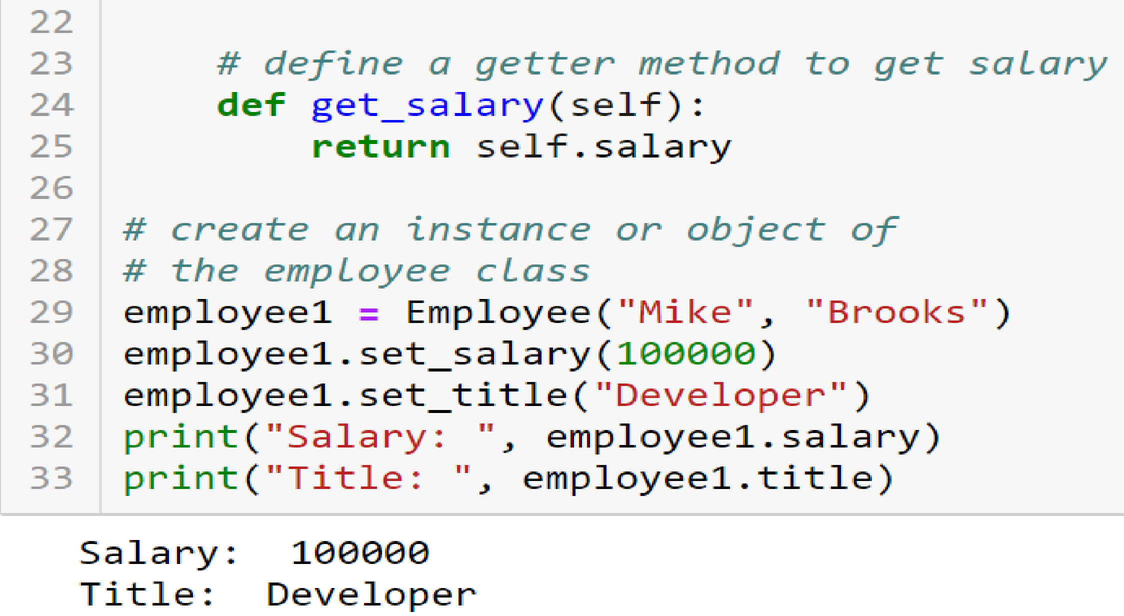
```
# some instance variables are parameters and some are not
def __init__(self, first_name="", last_name="", title="staff"):
    """initialize the instance variables"""
    self.first_name = first_name
    self.last_name = last_name
    self.title = title
    self.salary = None
```

- Here, some instance variables are parameters and some are not. The default values of instance variables can be changed using setter methods

Define the Setter Method



```python
# add setter methods to the employee class

class Employee():
    "Represents an employee"

    def __init__(self, first_name, last_name,
                 title="staff", salary=None):
        """initialize the instance variables"""

        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

    # define a setter method to set salary
    def set_salary(self, salary_amount):
        self.salary = salary_amount

    # define a setter method to set title
    def set_title(self, title):
        self.title = title
```



```python

    # define a getter method to get salary
    def get_salary(self):
        return self.salary

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.set_title("Developer")
print("Salary: ", employee1.salary)
print("Title: ", employee1.title)
```

```
Salary:  100000
Title:  Developer
```

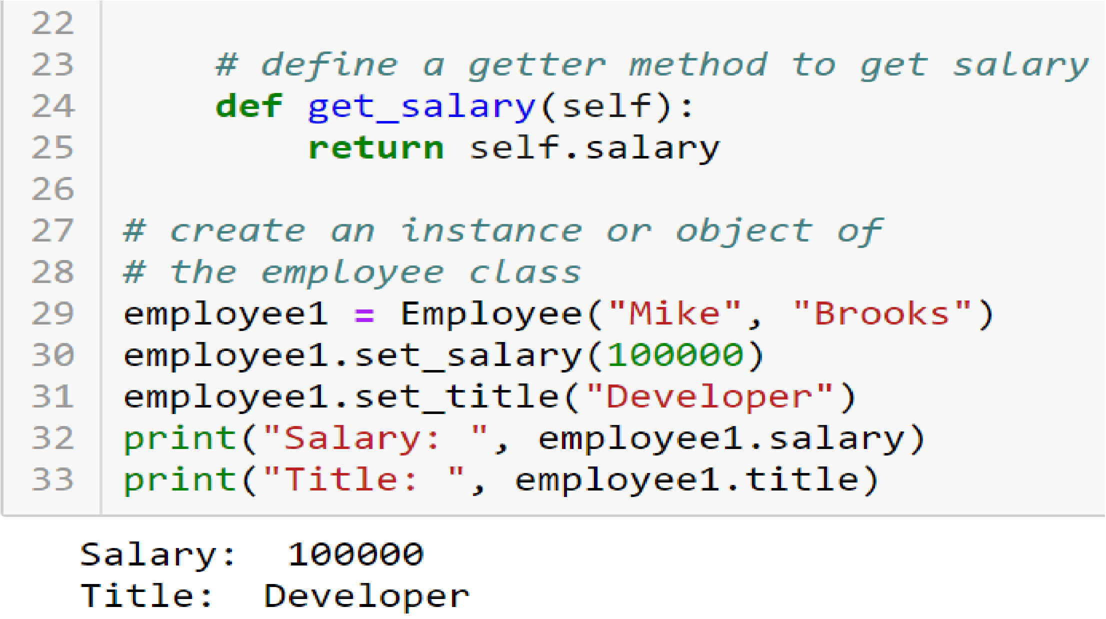
Setter or mutator methods are used to modify, change or update the values referenced by instance variables.

Using the dot Notation to Access Data in Objects

- The dot operator can be used to access data in objects.
- The general syntax for this is:

  object.attribute

- using the dot notation with an attribute is usually not the best option. It is better to rather use a getter or accessor method.

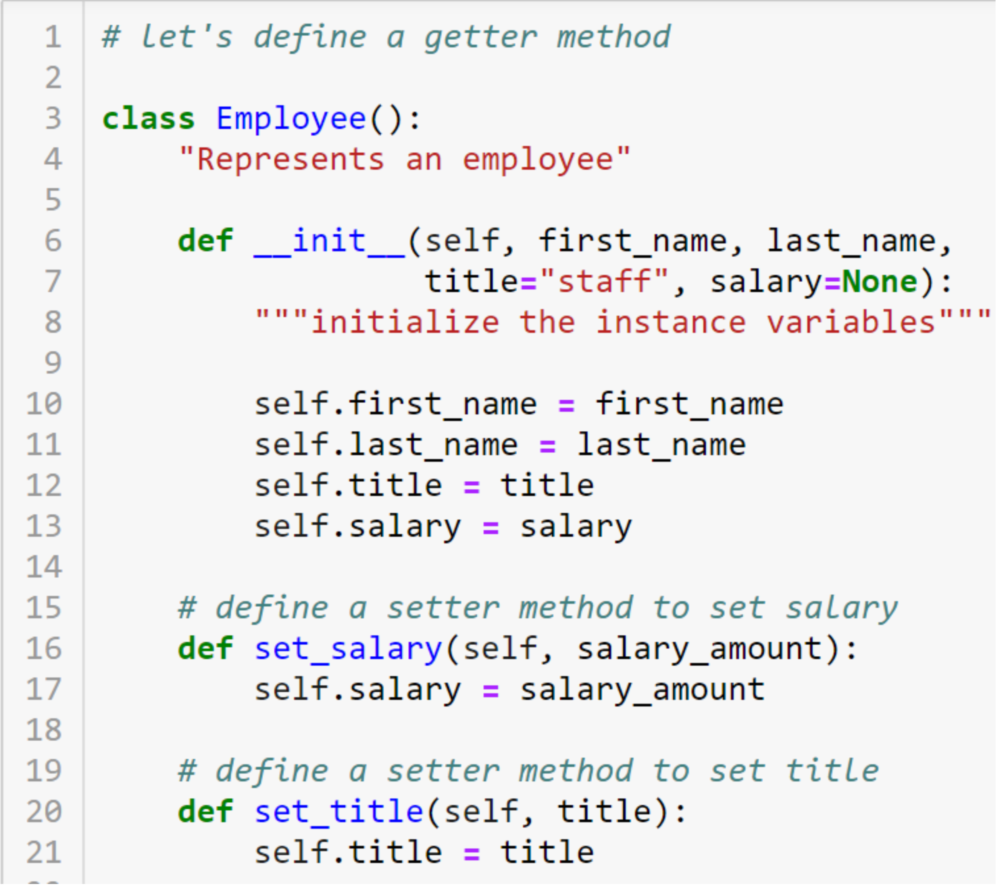


```

    # define a getter method to get salary
    def get_salary(self):
        return self.salary

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.set_title("Developer")
print("Salary: ", employee1.salary)
print("Title: ", employee1.title)
```

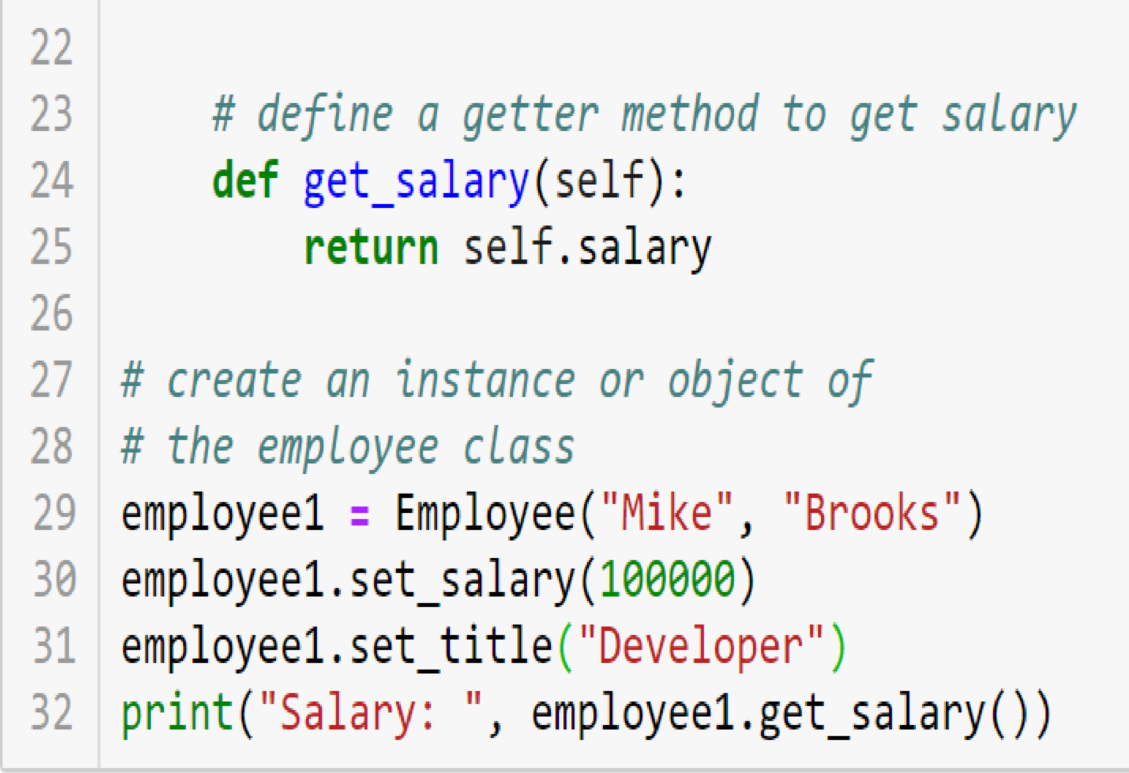
```
Salary:  100000
Title:  Developer
```

Here, we are using the dot notation with an attribute name to access the data.

Define the Getter Method



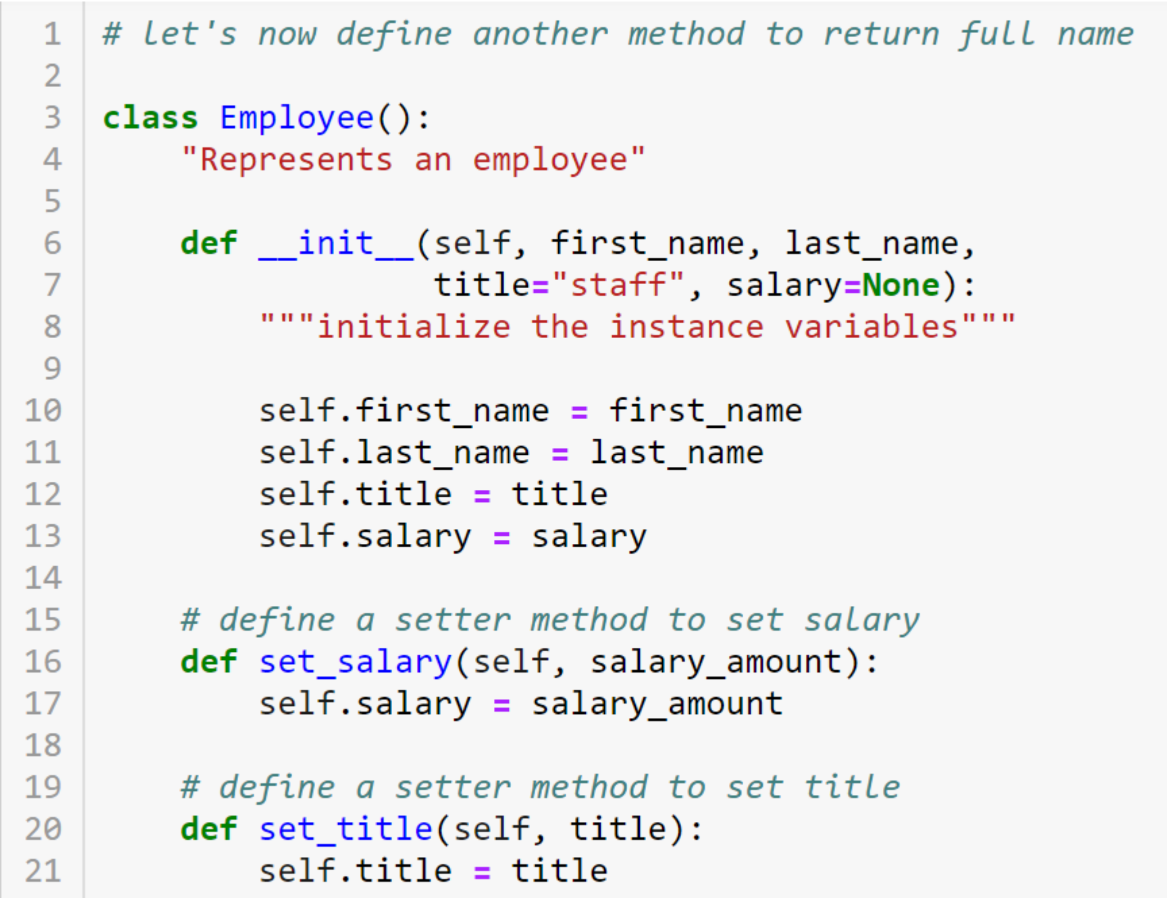
```python
# let's define a getter method

class Employee():
    "Represents an employee"

    def __init__(self, first_name, last_name,
                 title="staff", salary=None):
        """initialize the instance variables"""

        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

    # define a setter method to set salary
    def set_salary(self, salary_amount):
        self.salary = salary_amount

    # define a setter method to set title
    def set_title(self, title):
        self.title = title
```



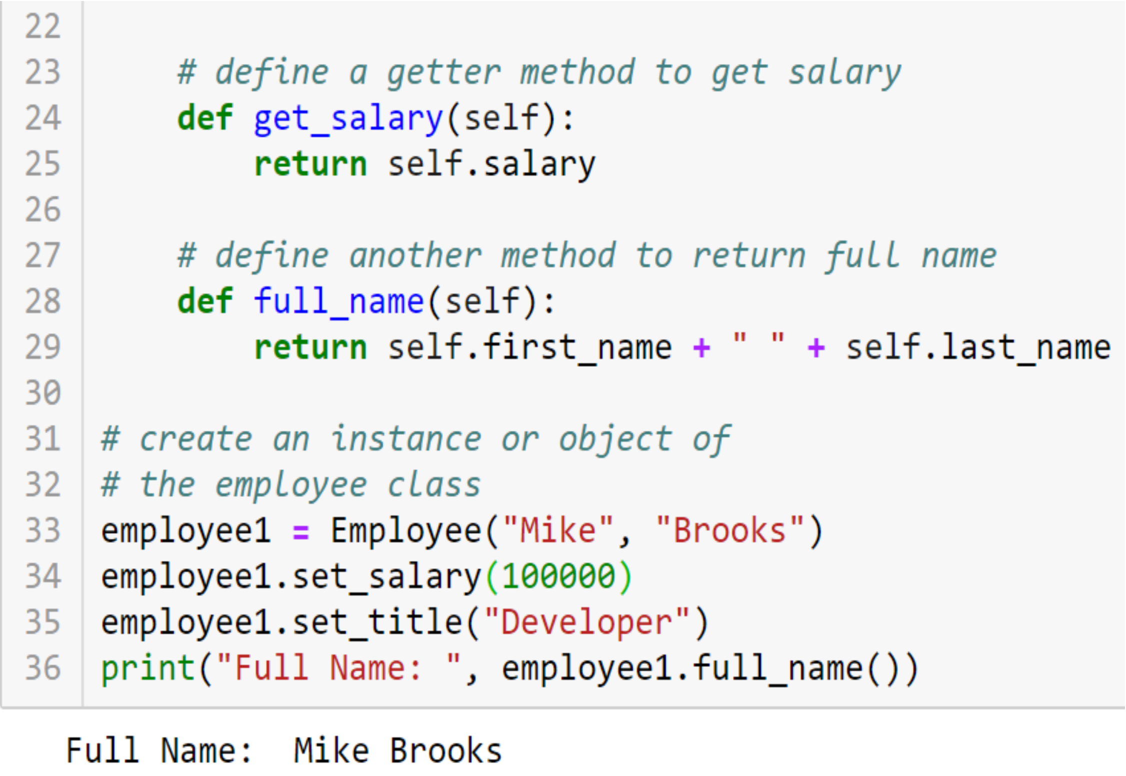
```python

    # define a getter method to get salary
    def get_salary(self):
        return self.salary

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.set_title("Developer")
print("Salary: ", employee1.get_salary())
```

```
Salary:  100000
```

Define Other Methods



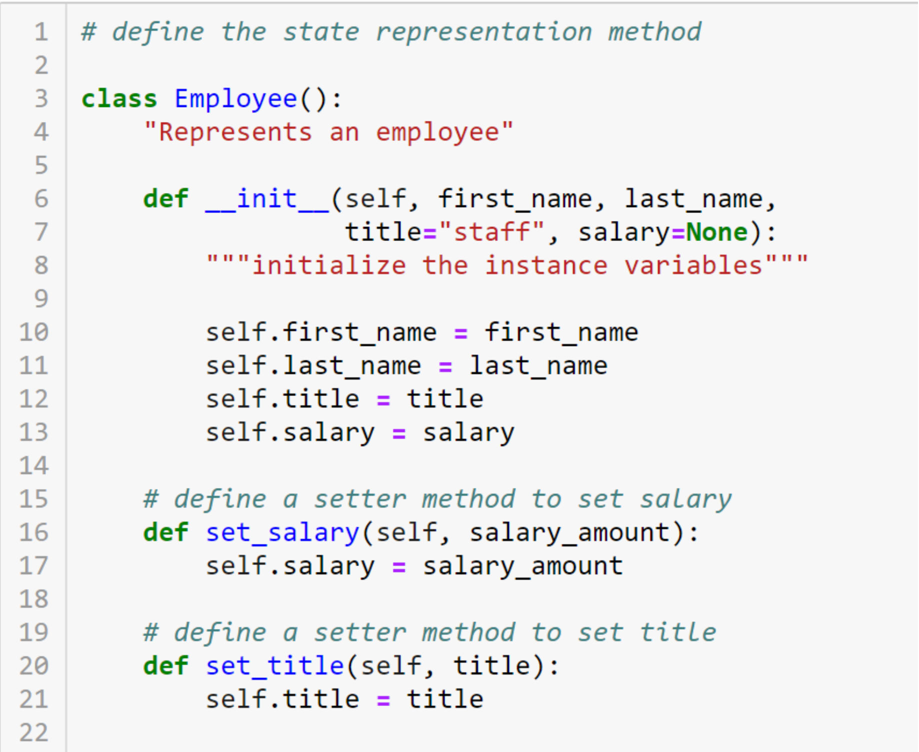
```python
# let's now define another method to return full name

class Employee():
    "Represents an employee"

    def __init__(self, first_name, last_name,
                 title="staff", salary=None):
        """initialize the instance variables"""

        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

    # define a setter method to set salary
    def set_salary(self, salary_amount):
        self.salary = salary_amount

    # define a setter method to set title
    def set_title(self, title):
        self.title = title
```



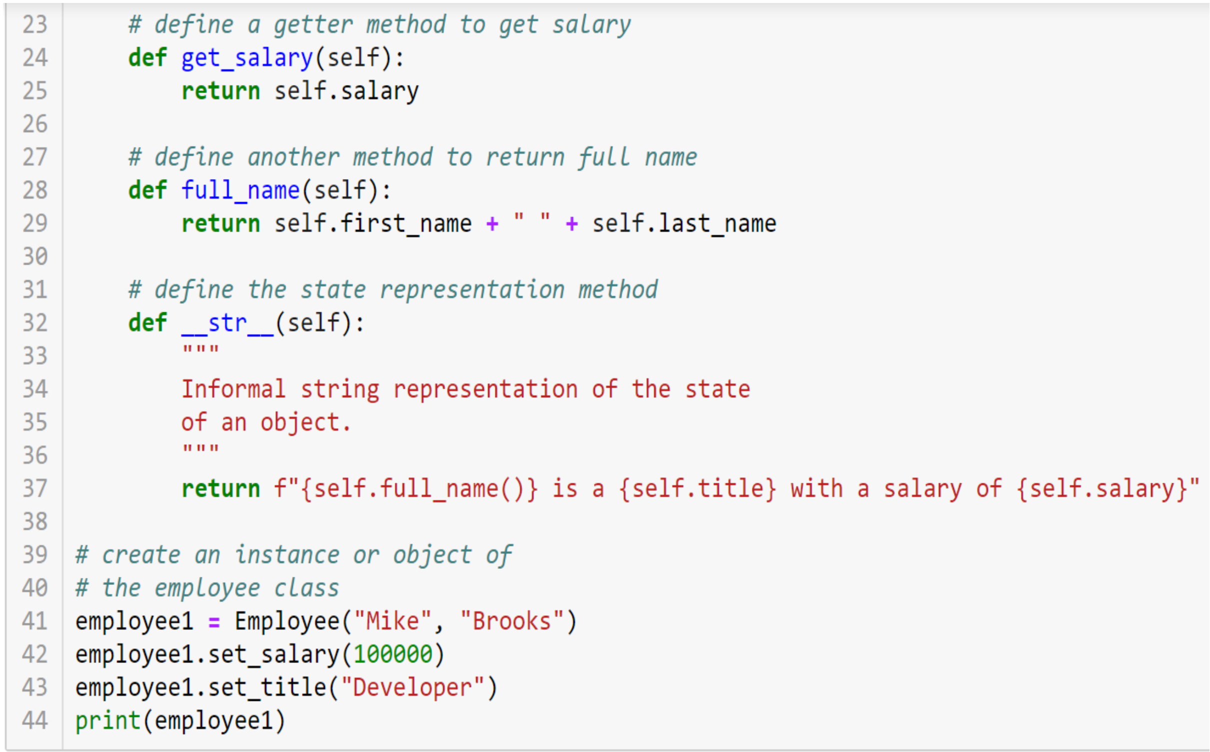
```python

    # define a getter method to get salary
    def get_salary(self):
        return self.salary

    # define another method to return full name
    def full_name(self):
        return self.first_name + " " + self.last_name

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.set_title("Developer")
print("Full Name: ", employee1.full_name())
```

```
Full Name:  Mike Brooks
```

Define the State Representation Method



```python
# define the state representation method

class Employee():
    "Represents an employee"

    def __init__(self, first_name, last_name,
                 title="staff", salary=None):
        """initialize the instance variables"""

        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

    # define a setter method to set salary
    def set_salary(self, salary_amount):
        self.salary = salary_amount

    # define a setter method to set title
    def set_title(self, title):
        self.title = title

```

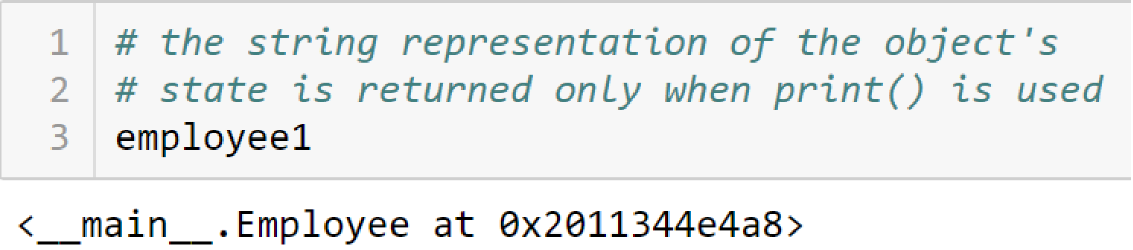


```python
    # define a getter method to get salary
    def get_salary(self):
        return self.salary

    # define another method to return full name
    def full_name(self):
        return self.first_name + " " + self.last_name

    # define the state representation method
    def __str__(self):
        """
        Informal string representation of the state
        of an object.
        """
        return f"{self.full_name()} is a {self.title} with a salary of {self.salary}"

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.set_title("Developer")
print(employee1)
```

```
Mike Brooks is a Developer with a salary of 100000
```

State Representation Method

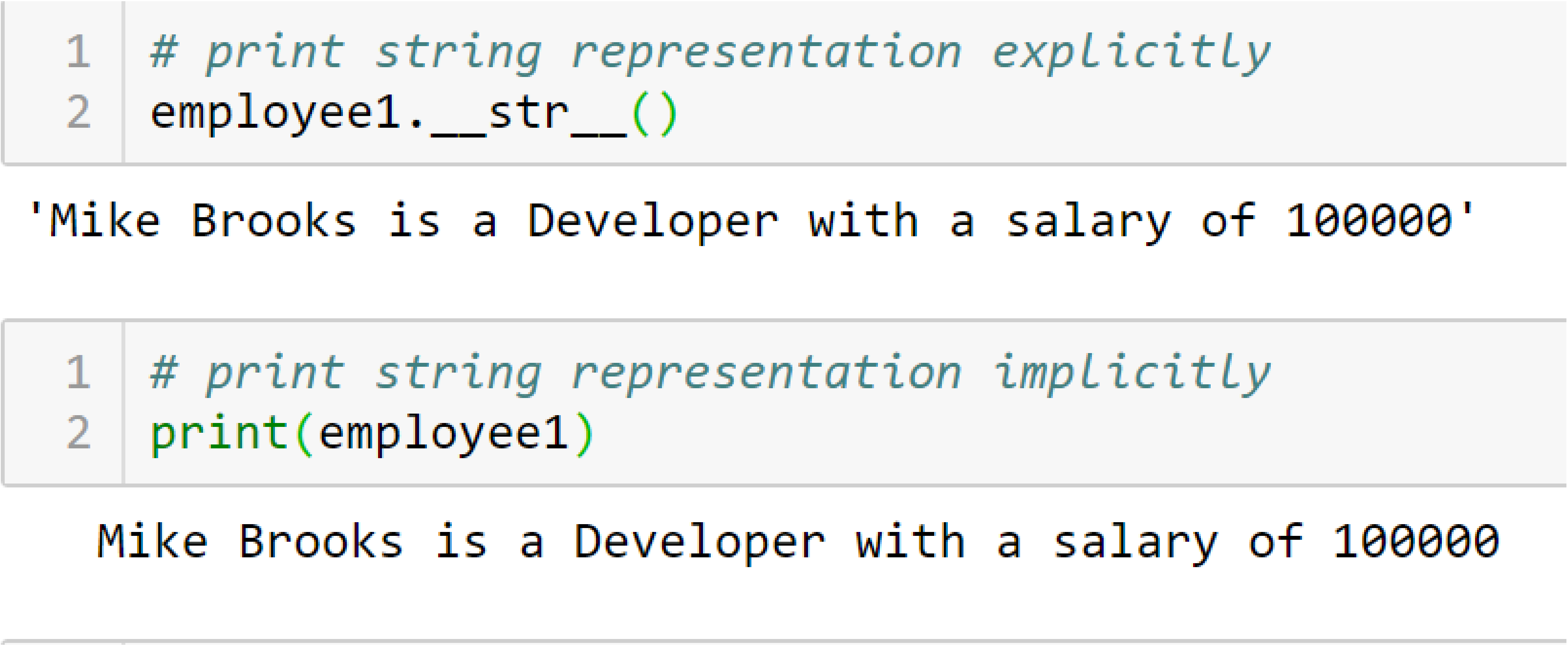
- The state representation method is the __str__() method. It provides an informal (human readable) string representation of the object’s state.
- The string representation of an object is returned when the instance of the class is created and printed.
- The string representation of the object would not be returned if a print() function is not used to print the instance object.



```
# the string representation of the object's
# state is returned only when print() is used
employee1
```

```
<__main__.Employee at 0x2011344e4a8>
```

How to Output String Representation



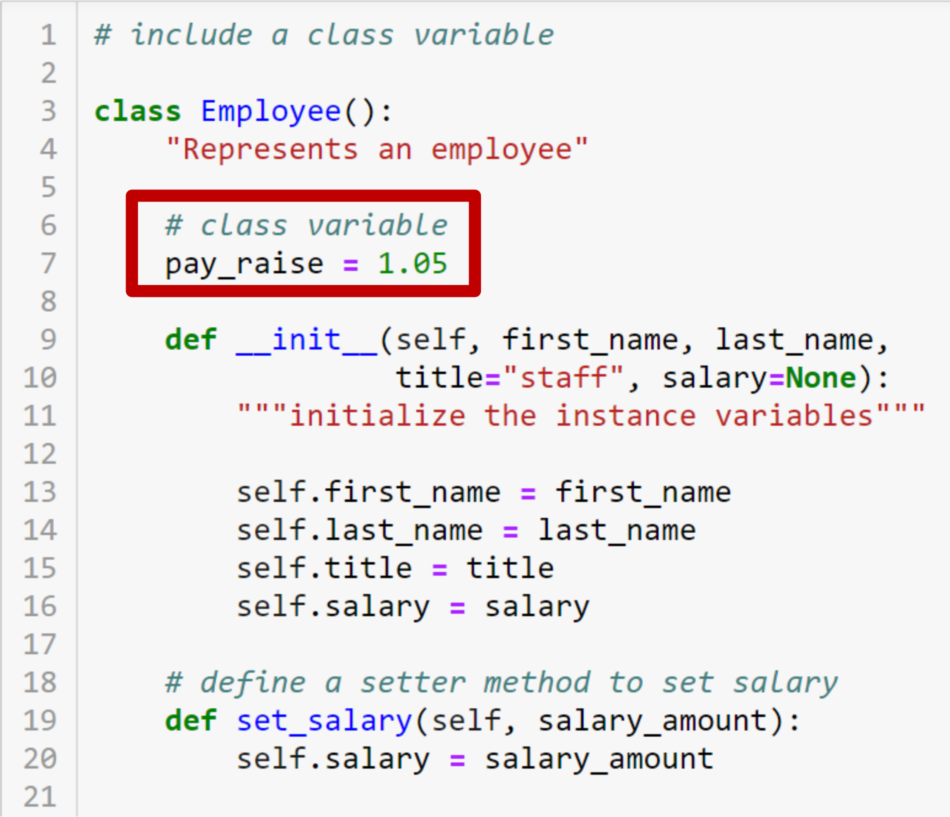
```python
# print string representation explicitly
employee1.__str__()
```

```
'Mike Brooks is a Developer with a salary of 100000'
```

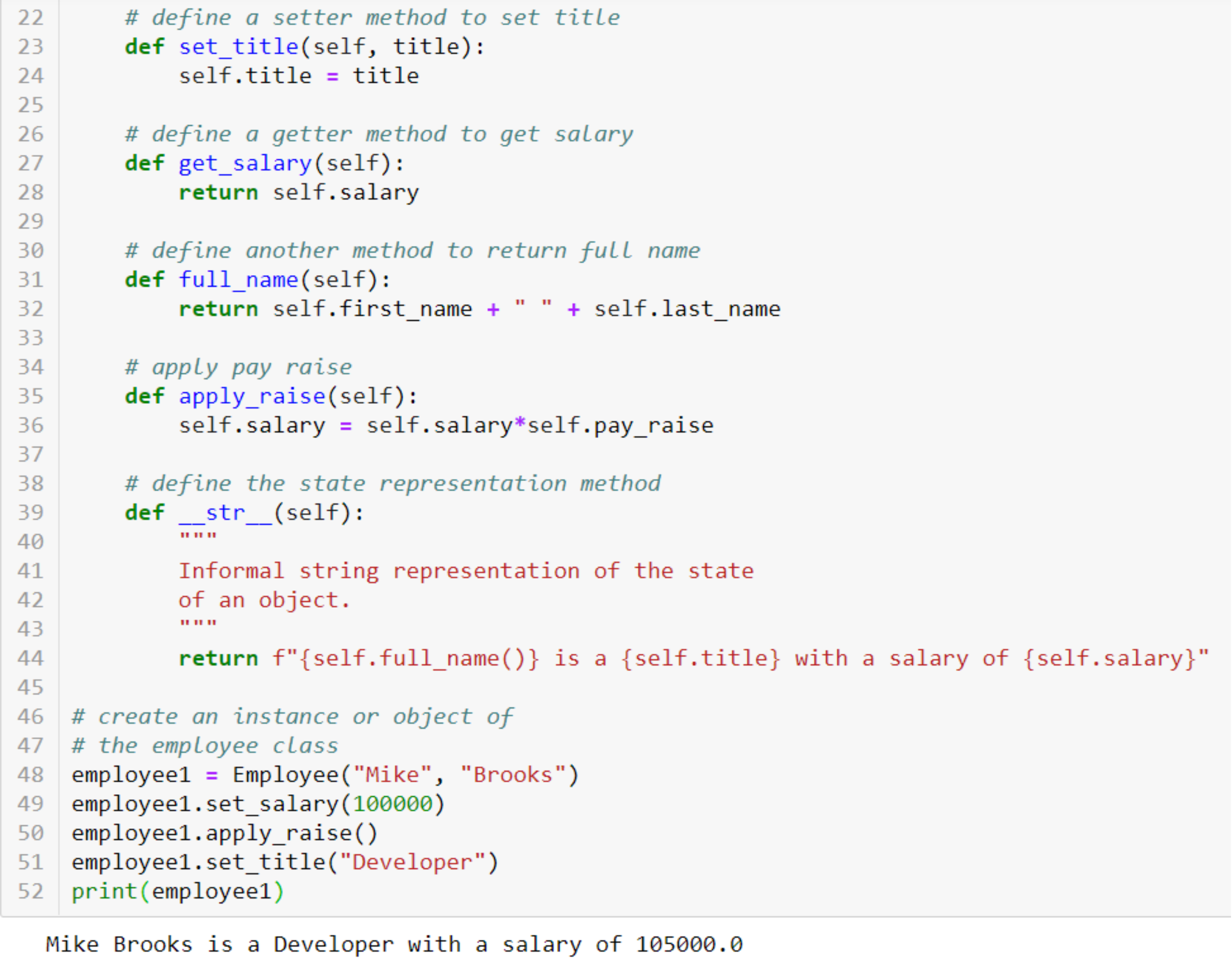
```python
# print string representation implicitly
print(employee1)
```

```
Mike Brooks is a Developer with a salary of 100000
```

Define a Class Variable



```python
# include a class variable

class Employee():
    "Represents an employee"

    # class variable
    pay_raise = 1.05

    def __init__(self, first_name, last_name,
                 title="staff", salary=None):
        """initialize the instance variables"""

        self.first_name = first_name
        self.last_name = last_name
        self.title = title
        self.salary = salary

    # define a setter method to set salary
    def set_salary(self, salary_amount):
        self.salary = salary_amount

```

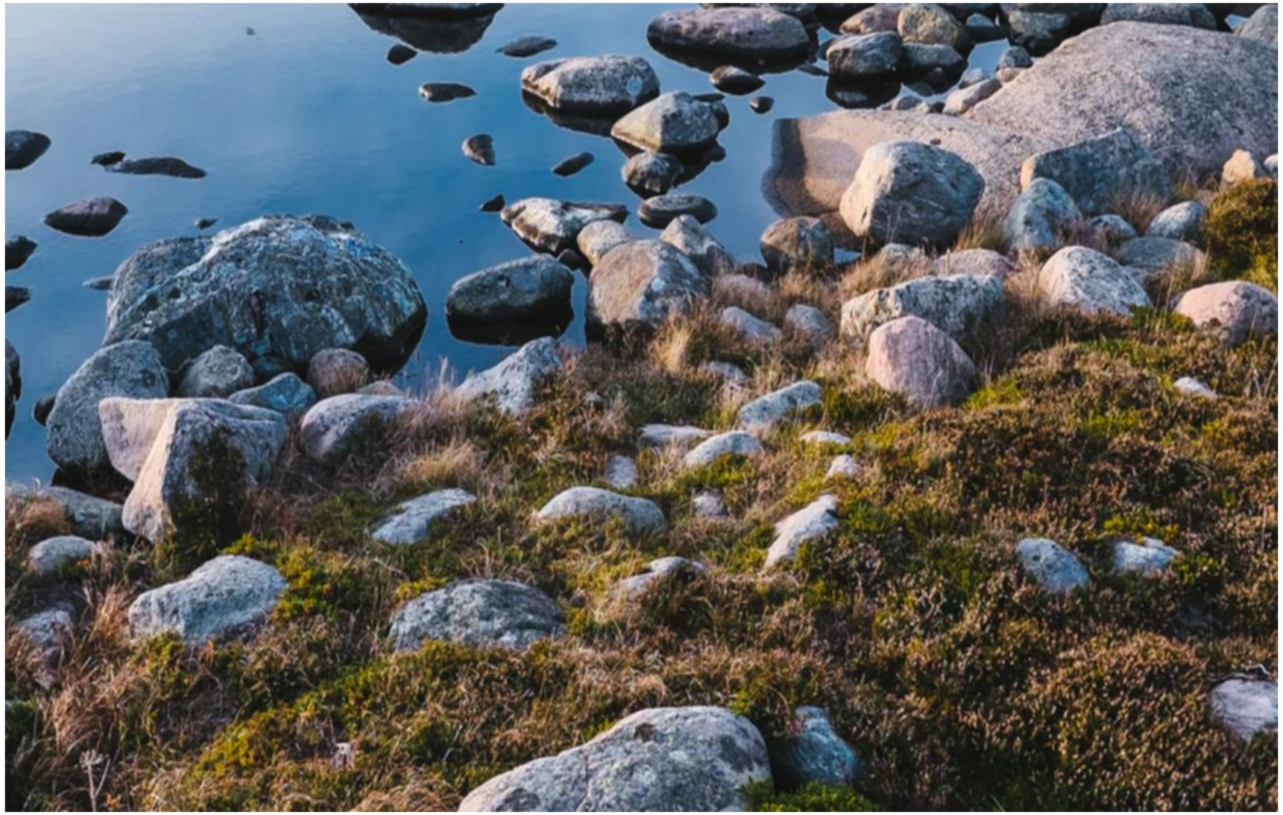


```python
    # define a setter method to set title
    def set_title(self, title):
        self.title = title

    # define a getter method to get salary
    def get_salary(self):
        return self.salary

    # define another method to return full name
    def full_name(self):
        return self.first_name + " " + self.last_name

    # apply pay raise
    def apply_raise(self):
        self.salary = self.salary*self.pay_raise

    # define the state representation method
    def __str__(self):
        """
        Informal string representation of the state
        of an object.
        """
        return f"{self.full_name()} is a {self.title} with a salary of {self.salary}"

# create an instance or object of
# the employee class
employee1 = Employee("Mike", "Brooks")
employee1.set_salary(100000)
employee1.apply_raise()
employee1.set_title("Developer")
print(employee1)
```

```
Mike Brooks is a Developer with a salary of 105000.0
```

Other OOP concepts

- Other concepts in OOP include:
  - abstraction,
  - encapsulation,
  - inheritance and
  - polymorphism



Congratulations!

- You have added another valuable programming skill to your skill set.
- At this point, we are ready to launch into data analysis

End of Lesson