**Classes and Objects in Python**

In this section we shall cover the core functions of Python objects and classes. We shall cover what a class is and how to create a class and also how to use a class in a Python program.

Python programming language is an Object Oriented Programming language (OOP). Unlike procedural programming languages where the main emphasis is on functions, OOP stresses its weight on objects. **Python** is an **object** oriented programming language. Almost everything in **Python** is an **object**, with its properties and methods. A **Class** is like an **object** constructor, or a "blueprint" for creating **objects**.

An object is a collection of data (variables) and methods (functions) which act on the data itself. Similarly, a class is a blueprint for the same object.

Think of a class as a sketch/draft (prototype) or a (model) of a house. The mode; contains all the details about the floors, doors, windows, plumbing, electrical circuitry, drainage etc. Now, based on these analogies, we build a real house. The house in this example is the object.

Remember, many real life houses can be made from a house blueprint, we can create many objects from a class. An object is also an instance of a class and the process of creating this object is called **instantiation**.

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects.

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or pre-processed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

For instance, an object could represent a person with properties like a **name**, **age**, **height**, **email address** , **phone number** and **address** and behaviors such as **walking**, **talking**, **breathing**, and **running**.

Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relationships between things, like companies and employees, students and teachers, and so on.

OOP models real-world entities (attributes) as software objects that have some data associated with them and can perform certain functions.

Another programming paradigm we talked about earlier is procedural programming, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, that flow sequentially in order to complete a task.

The key takeaway here is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

## **Define a Class in Python**

## Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, let’s say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

**Monicah = [“Monicah Booker”, 31, “Sale Executive”, 3423]**

**Jorge = [“Jorge Ramos”, 30, “Web Developer”, 3983]**

**Chantelle = [“Chantelle Abrams”, “Graphics Designer”, 3933]**

Looking at above list items, you will quickly realize that there are a number of issues with this type of approach.

First, it can make larger code files more difficult to navigate and manage. If you reference **Monicah[0]** several lines away from where the **Monicah** list is declared, will you remember that the element with index 0 is the employee’s name?

Secondly, it can introduce errors if not every employee has the same number of elements in the list. In the **Chantelle** list above, the age is missing, so **Chantelle[1]** will return "**Graphics Designer**" instead of **Chantelle**’s age.

A great way to make this type of code more manageable and more maintainable is to use classes.

### **Classes vs Instances**

### Classes are used to create user-defined data structures. Classes define functions called methods, which identify the behaviors and actions that an object created from the class can perform with its data.

In this section, we’ll create a Dog class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how something should be defined with its attributes. It doesn’t actually contain any data. The Dog class specifies that a name, breed and an age are necessary for defining a dog, but it doesn’t contain the name or age of any specific dog in particular.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It’s an actual dog with a name, breed, like Jack, a german Shepherd, and who’s four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information. Just like many people can fill out the same form with their own unique identifying information, many instances can be created from a single class.

## **Constructors in Python**

Class functions that begin with double underscore \_\_ are called special functions as they have special meaning.

Of one particular interest is the \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

## **Defining a Class in Python**

Like function definitions in Python begin with the **def** keyword, class definitions begin with a class keyword.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

**class MyNewClass:**

**'''This is a docstring. I have created a new class'''**

**Pass**

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores \_\_. For example, \_\_doc\_\_ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class’s body.

**Class George:**

**‘’’ This is docstring. This is a new class created’’’**

**pass**

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

**Note:** Python class names are written in CapitalizedWords notation by convention. For example, a class for a specific breed of dog like the German Shepard would be written as GermanShepard.

The Dog class right now does not have much yet, so let’s spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed.

The properties that all Dog objects must have are defined in a method called .\_\_init\_\_(). Every time a new Dog object is created, .\_\_init\_\_() sets the initial state of the object by assigning the values of the object’s properties. That is, .\_\_init\_\_() initializes each new instance of the class.

You can give .\_\_init\_\_() any number of parameters, but the first parameter will always be a variable called self. When a new class instance is created, the instance is automatically passed to the self parameter in .\_\_init\_\_() so that new attributes can be defined on the object.

Let’s update the Dog class with an .\_\_init\_\_() method that creates .name and .age attributes:

**Python Class declaration example**

**class Dog:**

**def \_\_init\_\_(self, name, breed, color, age):**

**self.name = Jimmy**

**self.breed = German Shepard**

**self.color = black**

**self.age = age**

Notice that the .\_\_init\_\_() method’s signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the .\_\_init\_\_() method belongs to the Dog class.

In the body of .\_\_init\_\_(), there are four statements using the dog variable:

1. self.name = name creates an attribute called name and assigns to it the value of the name parameter.
2. self.breed = breed creates an attribute called breed and assigns to it the value of the breed parameter.
3. self.color = color creates an attribute called color and assigns to it the value of the color parameter.
4. self.age = age creates an attribute called age and assigns to it the value of the age parameter.

Attributes created in .\_\_init\_\_() are called instance attributes. An instance attribute’s value is specific to a particular instance of the class. All Dog objects have a name, breed, color coat and an age, but the values for the name, breed, color coat and age attributes will vary depending on the Dog instance.

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .\_\_init\_\_().

For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

class Dog:

# Class attribute

species = "Canis familiaris"

def \_\_init\_\_(self, name, breed, color, age):

self.name = name

self.breed = breed

self.color = color

self.age = age

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a Dog class, let’s create some dogs!

### Class and Instance Attributes

Now create a new Dog class with a class attribute called .species and four instance attributes called .name, .breed, color and .age:

>>> class Dog:

... species = "Canis familiaris"

... def \_\_init\_\_(self, name, breed, color, age):

... self.name = name

... self.breed = breed

... self.color = color

... self.age = age

To instantiate objects of this Dog class, you need to provide values for the name, breed, color coat and age. If you don’t, then Python raises a TypeError:

>>> Dog()

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

Dog()

TypeError: \_\_init\_\_() missing 4 required positional arguments: 'name', 'breed','color' and 'age'

To pass arguments to the name, breed, color and age parameters, put values into those parentheses after the class name:

**Typical Python Example code**

>>> Jimmy = Dog("Jimmy","Golden Retriever","Light Golden", 6)

>>> Jack = Dog("Jack","german shepherd","Black & Tan", 4)

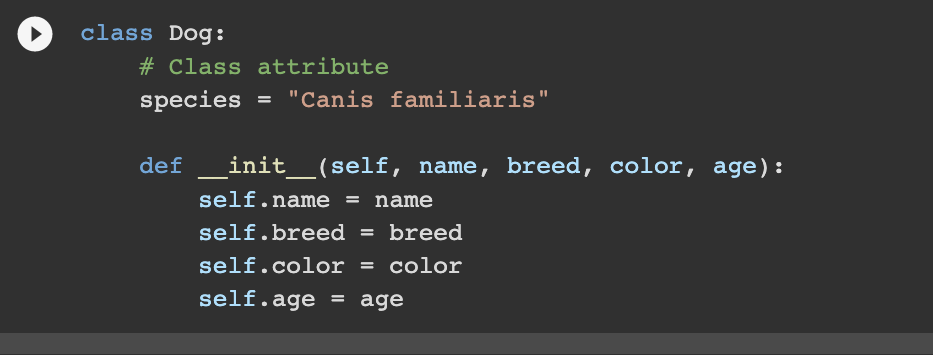
>>> Tobi = Dog("Tobi","cocker spaniel","Black", 2)

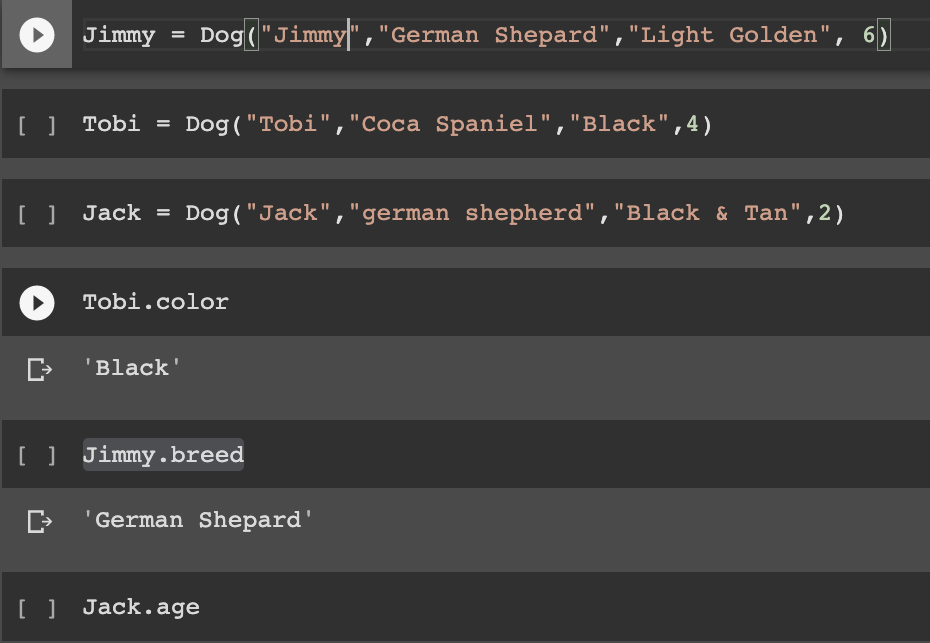
This creates three new Dog instances—one for a six-year-old dog named Jimmy, another for a dog named Jack four-year-old and another one for a two-year-old dog named Tobi.

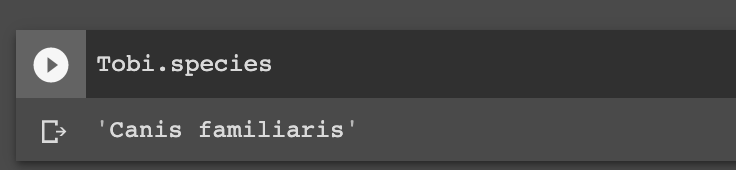
The Dog class’s .\_\_init\_\_() method has five parameters, so why are only four arguments passed to it in the example?

When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of .\_\_init\_\_(). This essentially removes the self parameter, so you only need to worry about the name, breed, color coat and age parameters.

After you create the Dog instances, you can access their instance attributes using dot notation:







One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All Dog instances have .species, .name, breed, color and .age attributes, so you can use those attributes with confidence knowing that they will always return a value.

Although the attributes are guaranteed to exist, their values *can* be changed dynamically:

>>> Tobi.age = 3

>>> Jack.age

4

>>> Jimmy.species = "Felis silvestris"

>>> Jimmy.species

'Felis silvestris'

In this example, you changed the .age attribute of the Tobi object to 3. Then we changed the .species attribute of the Jimmy object to "Felis silvestris", which is a species of cat. That makes Jimmy a pretty weird dog, but that is still valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

### **Instance Methods**

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like .\_\_init\_\_(), an instance method’s first parameter is always self.

Open a new editor window in IDLE and type in the following Dog class:

class Dog:

species = "Canis familiaris"

def \_\_init\_\_(self, name, age):

self.name = name

self.breed = breed

self.color = color

self.age = age

# Instance method

def description(self): #1

return f"{self.name} is {self.age} years old"

# Another instance method

def speak(self, sound):#2

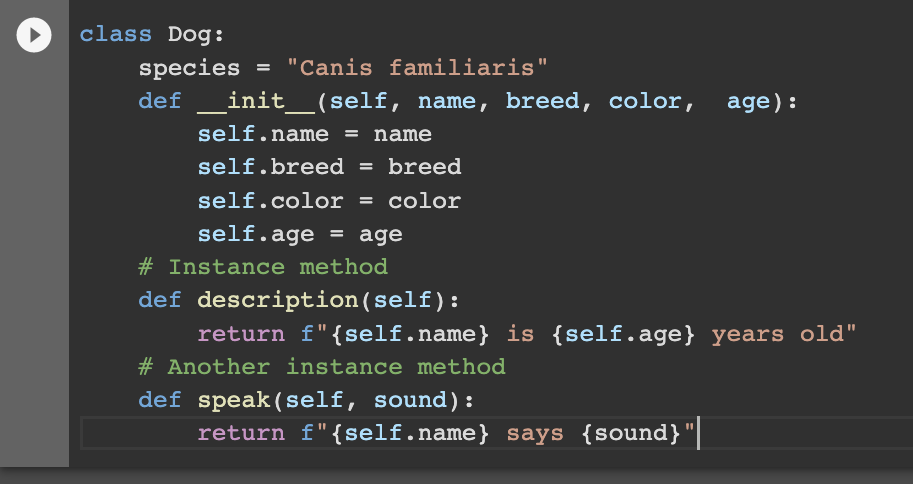
return f"{self.name} says {sound}"

**This Dog class has two instance methods:**

1. .description() returns a string displaying the name and age of the dog.
2. .speak() has one parameter called sound and returns a string containing the dog’s name and the sound the dog makes.

Save the modified Dog class to a file called dog.py and press F5 to run the program.

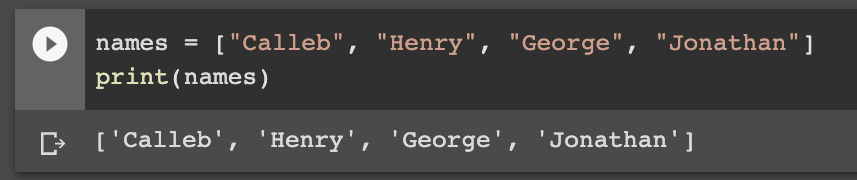
Then open the interactive window and type the following to see your instance methods in action:



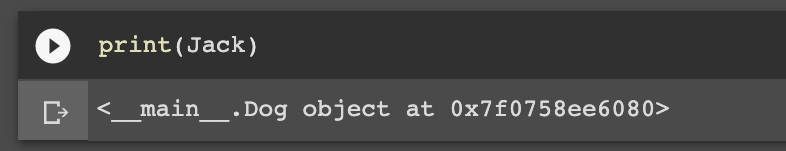


In the above Dog class, .description() returns a string containing information about the Dog instance Jack. When writing your own classes, it’s a good idea to have a method that returns a string containing useful information about an instance of the class. However, .description() isn’t the best way of doing this.

When you create a list object, you can use print() to display a string that looks like the list below:

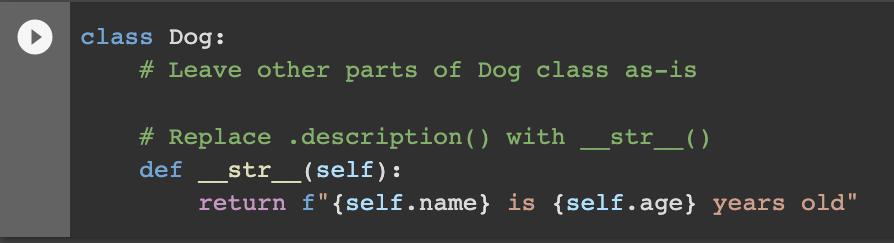


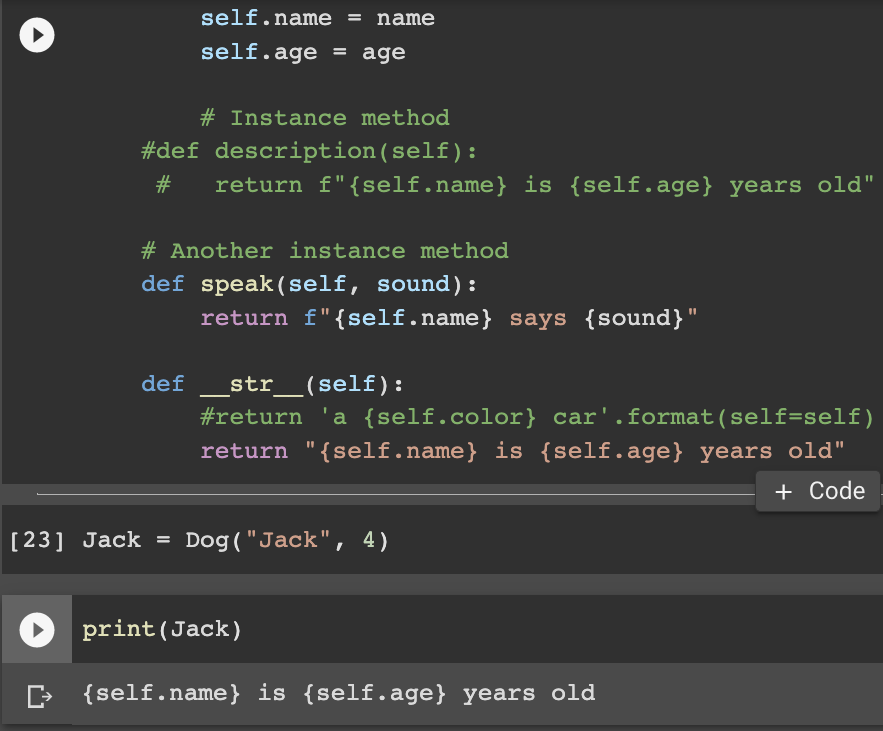
Let’s see what happens when you print() the miles object:



When you print(Jack), you get a cryptic looking message telling you that Jack is a Dog object at the memory address 0x7f0758ee6080. This message isn’t very helpful. You can change what gets printed by defining a special instance method called .\_\_str\_\_().

In the editor window, change the name of the Dog class’s .description() method to .\_\_str\_\_():



Save the file and press F5. Now, when you print(miles), you get a much friendlier output:

Methods like .\_\_init\_\_() and .\_\_str\_\_() are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that we can use to customize classes in Python. Although too advanced a topic for a beginning Python session, understanding **dunder methods** is an important part of mastering object-oriented programming in Python.

class ComplexNumber:

def \_\_init\_\_(self, r=0, i=0):

self.real = r

self.imag = i

def get\_data(self):

print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object

num1 = ComplexNumber(2, 3)

# Call get\_data() method

# Output: 2+3j

num1.get\_data()

# Create another ComplexNumber object

# and create a new attribute 'attr'

num2 = ComplexNumber(5)

num2.attr = 10

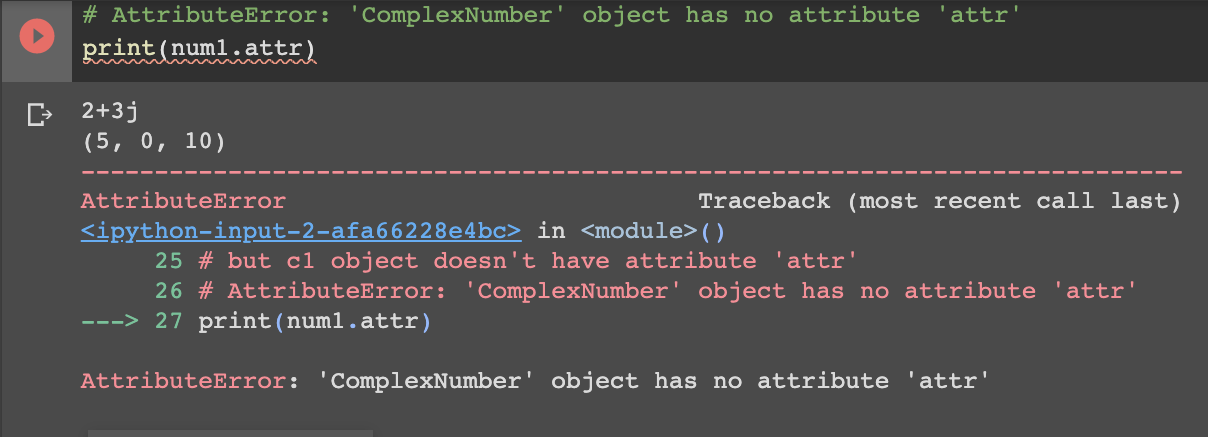
# Output: (5, 0, 10)

print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'

# AttributeError: 'ComplexNumber' object has no attribute 'attr'

print(num1.attr)



In the above example, we defined a new class to represent complex numbers. It has two functions, \_\_init\_\_() to initialize the variables (defaults to zero) and get\_data() to display the number properly.

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object num2 and read it as well. But this does not create that attribute for object num1.

**Points to take away**

To understand the meaning of classes we have to understand the built-in \_\_init\_\_() function. All classes have a function called \_\_init\_\_(), which is always executed when the class is being initiated.

Use the \_\_init\_\_() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

For brevity, let’s create another class to illustrate a real life scenario.

### 

### 

### 

### 

### 

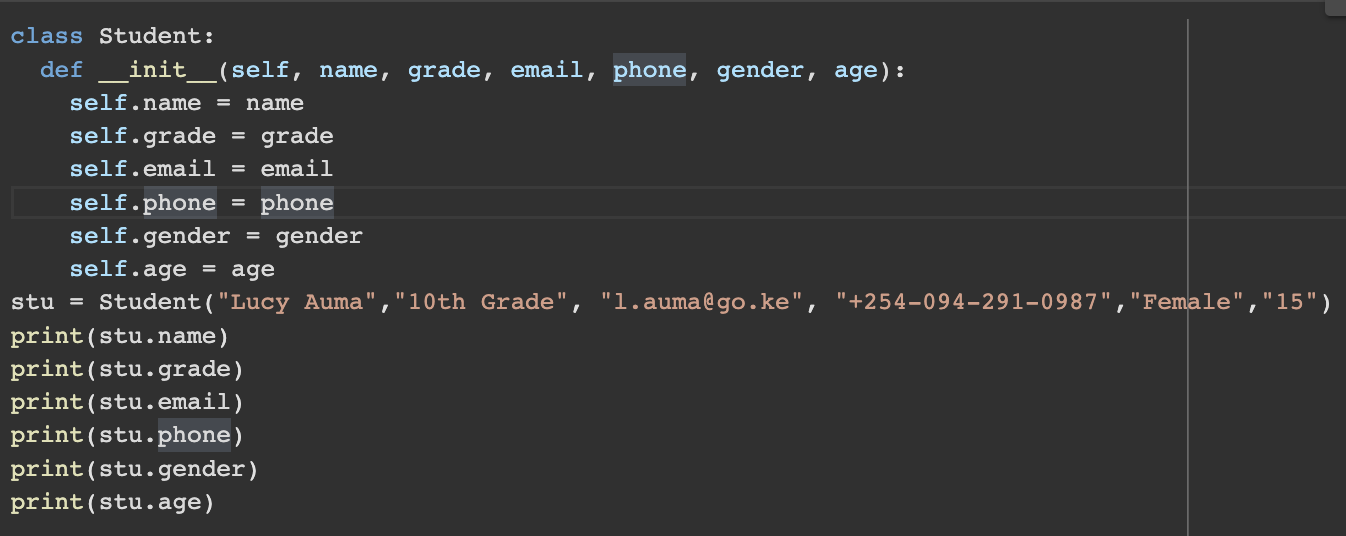
### 

### 

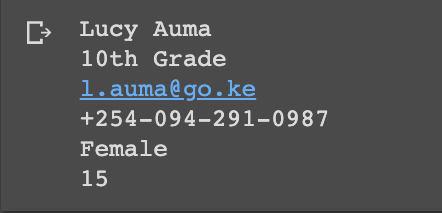
### 

### **Example**

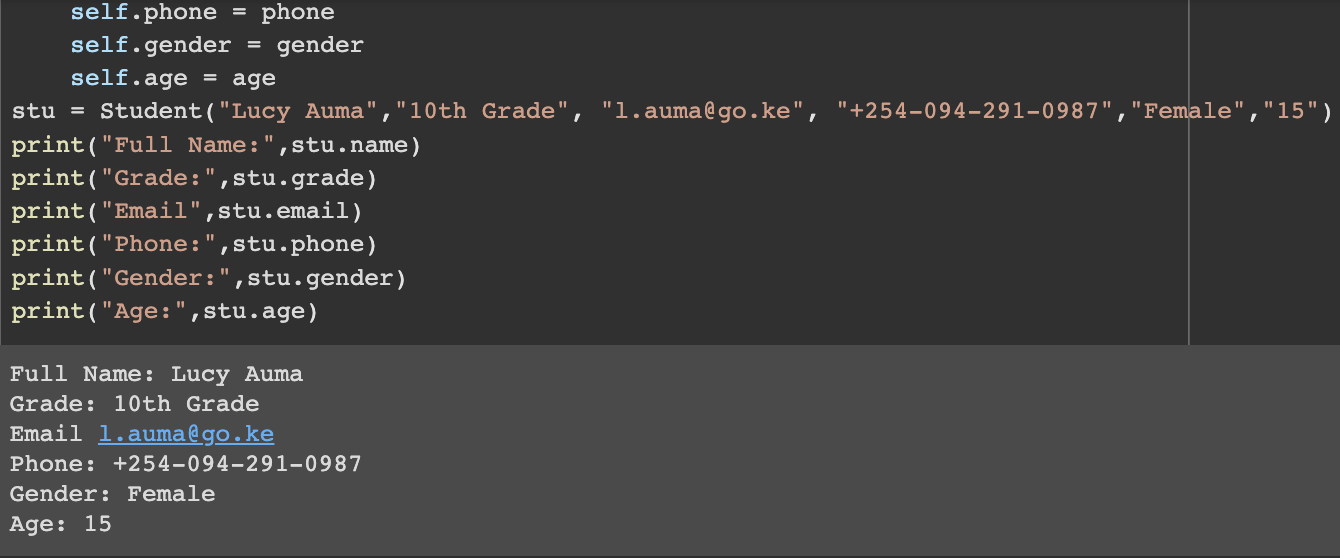
Create a class named student, use the \_\_init\_\_() function to assign values for name, grade, email, phone, gender and age:



Output of the Program when ran



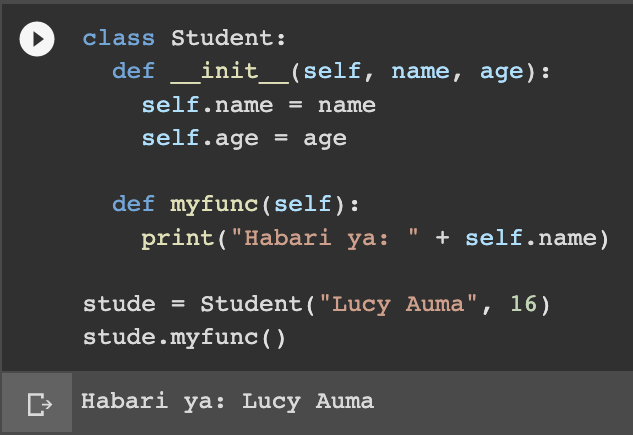
Or



**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

## Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object. Now create a method in the student class:



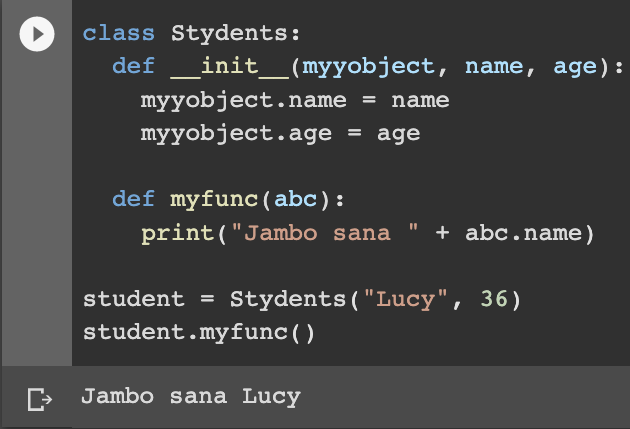
**Note:** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

## **The self Parameter**

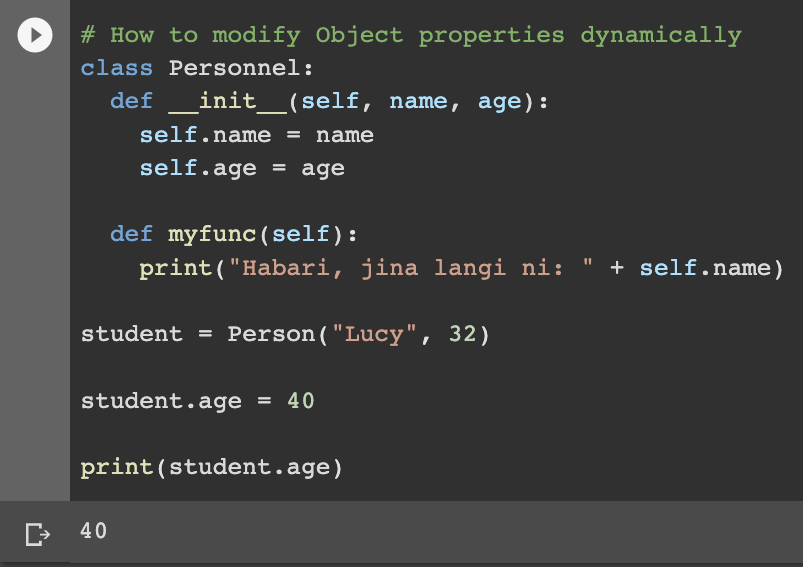
The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

It does not have to be named self , we can call it whatever we like, but it has to be the first parameter of any function in the class:

Using something else other than the word self

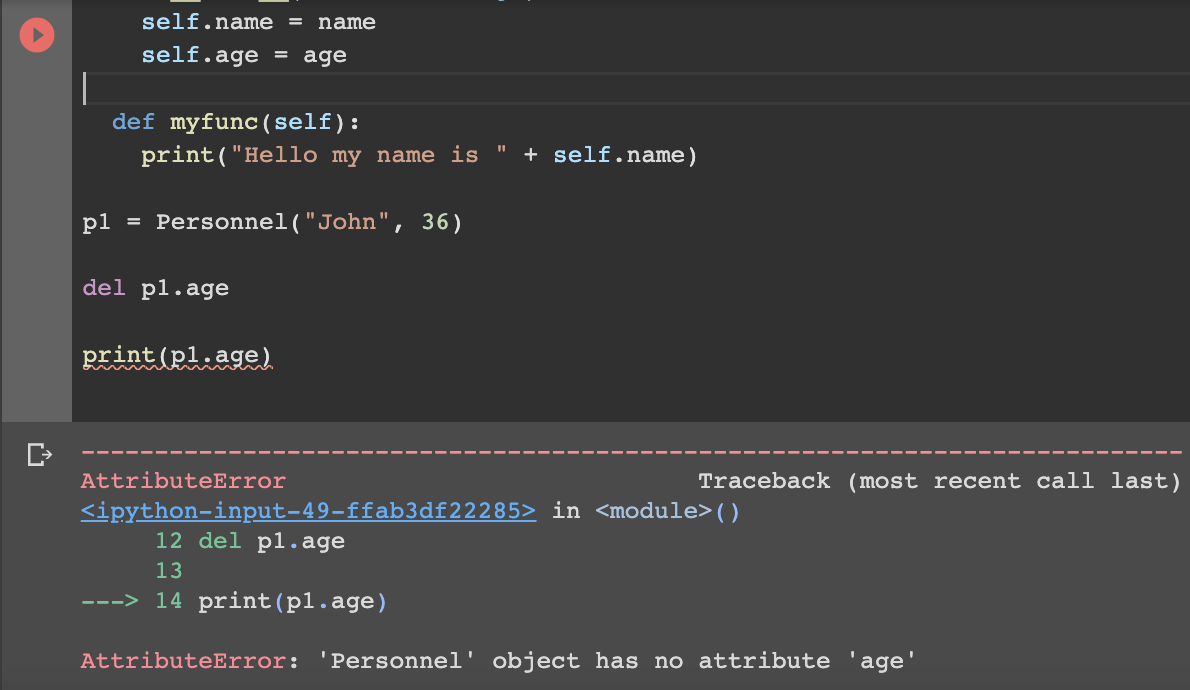


Next screen show how we can modify Object properties dynamically



The output will change. Instead of see 32. We shall get the new age to be 40 as the new assignment to student.age.

How we can delete Object properties. We can achieve this by using the del keyword.



## The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

### Example

class Personnel:

Pass

# having an empty class definition like this, would raise an error without the pass statement

**Evaluating your understanding of Classes**

Create a house class with 5 instance attributes:

1. .Type, this stores a type of the house as a string (bungalow, apartment, family)
2. .age, this stores how old is the house
3. .material, used for building the house as in (brick, board, wood)
4. .roof, type of roof for the house. As in is it metal, tile, corrugated sheets)
5. .color, color of the house. As in is it red, blue, pink, yellow, or what?)

Then instantiate two house objects—a house type house with 15 years and a house with 20 years —and print out the materials used, age, type and roof type. Your output should look like this:

The red house is 15 years old - It has Corrugated roof top, It is a bungalow unit.

The pink house is 20 years old - It has Corrugated roof top, It is a Family unit.