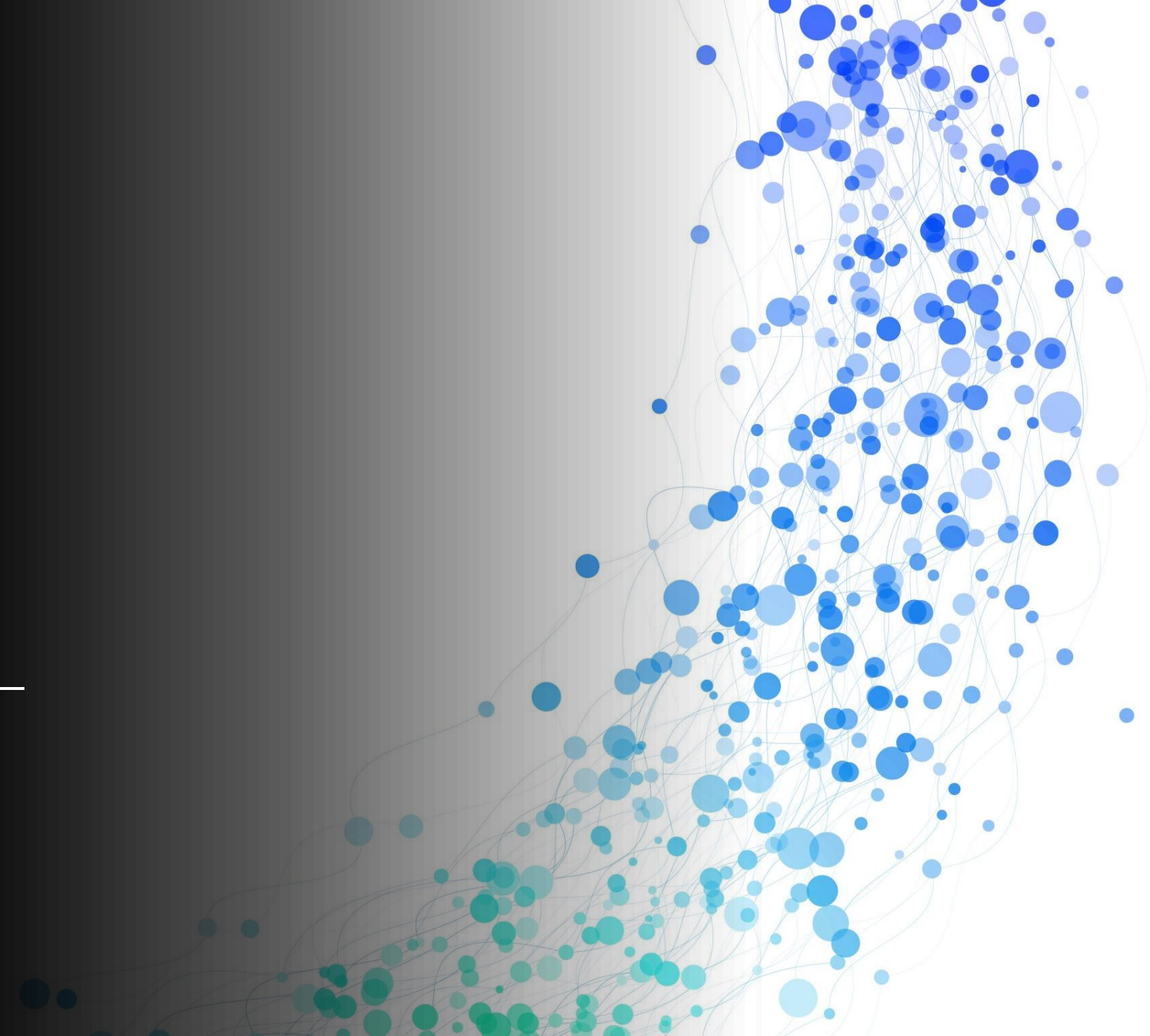
Object Oriented Python

Classes and Objects



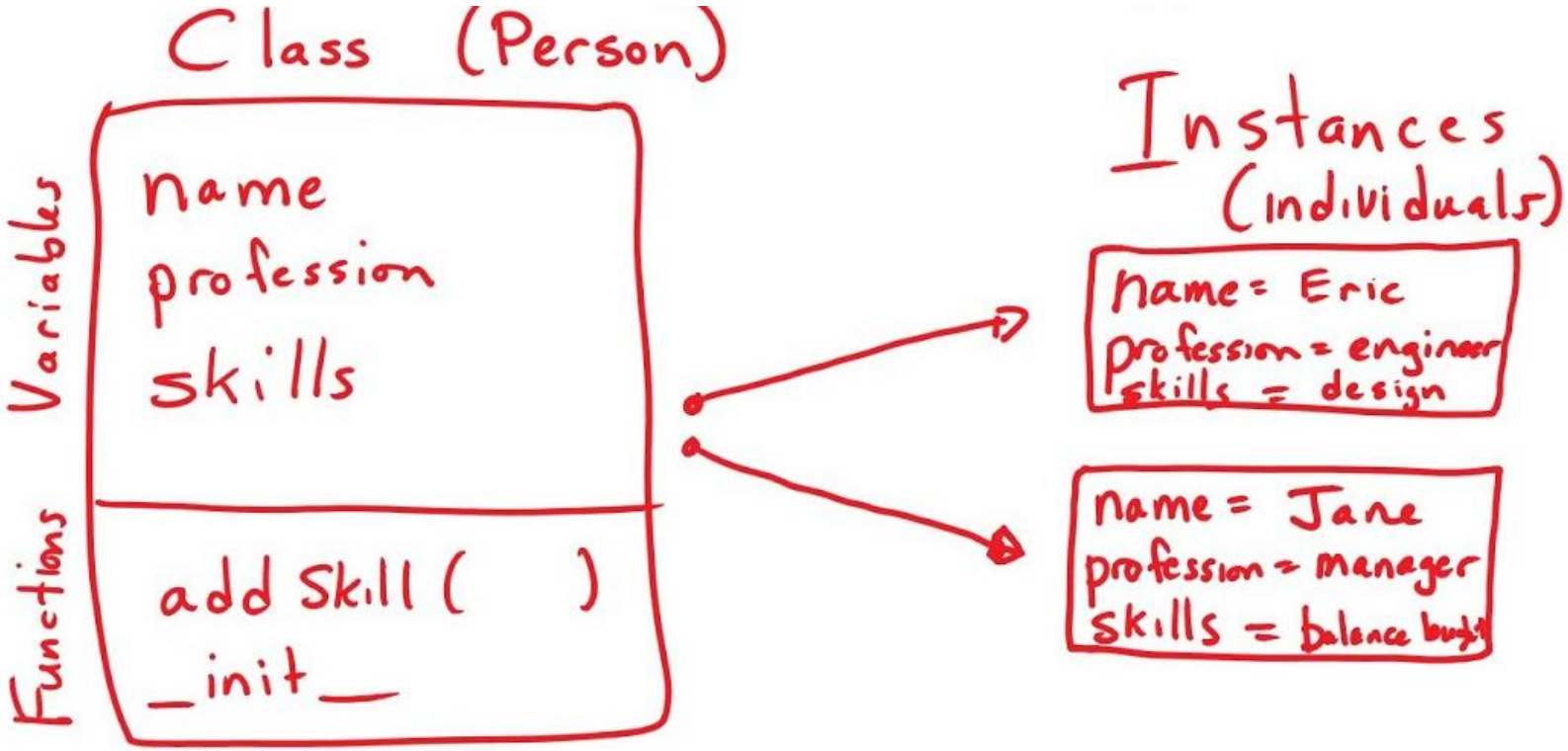
Introduction

- 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.

Classes

- A class is a user-defined blueprint or prototype from which objects are created.
- They provide a means of bundling data and functionality together.
- Creating a new class creates a new type of object, allow new instances of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

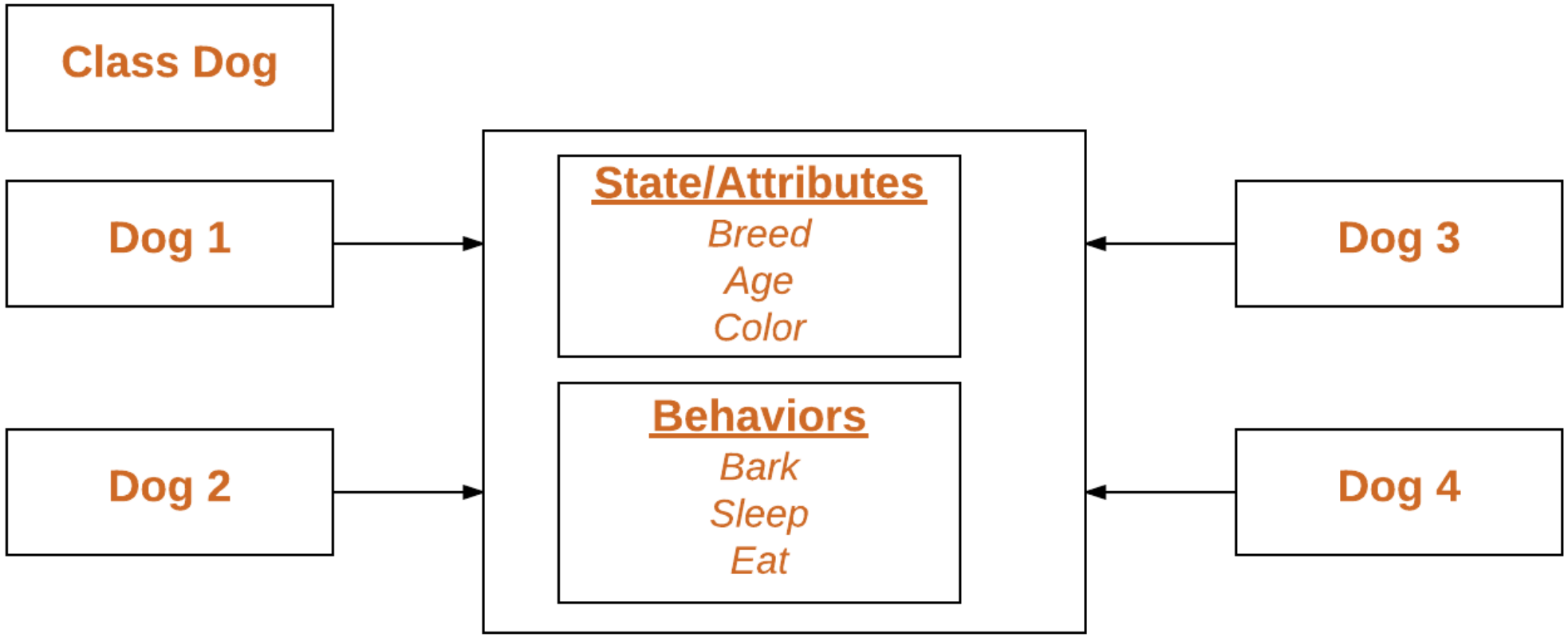
Class and Instances



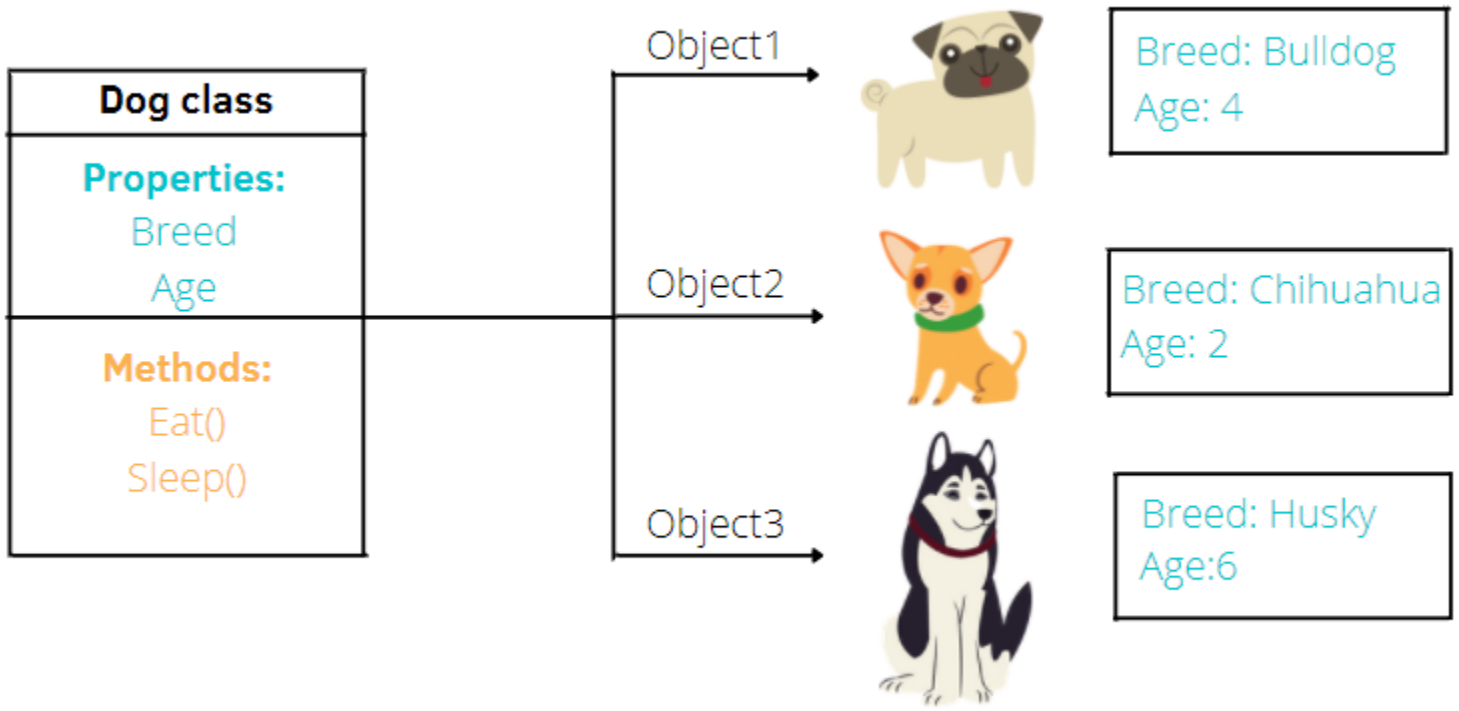
Why create Classes?

- To understand the need for creating a class in Python let’s consider an example:
- Let’s say you wanted to track the number of ***dogs*** that may have different attributes like **breed**, and **age**. If a list is used, the first element could be the dog’s breed while the second element could represent its age.
- Let’s suppose there are **100 different dogs**, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This **lacks organization** and it’s the exact need for classes.

Structure of a Class and its instances



Classes and Instances



Naming

- Objects/Instances
- **Attributes**/Field Names/Properties/Data Members

Structure of a Class in Code

attributes:

- speed
- gear

behaviours:

- speed up
- apply brake
- change gear

```
class bicycle:
    ''' properties'''
    # Class variables.
    gear = 1
    speed = 0

    def __init__(self, gear, speed):
        self.gear = gear
        self.speed = speed

    def speedUp(self, increase):
        self.speed += increase

    def changeGear(self,newGear):
        self.gear = newGear

    def applyBrake(self, decrease):
        self.speed -= decrease
```

Creating a Class in Python

To create a class, use the keyword `class`:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Creating an Object in Python

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Class example

```
class User:
    pass
```

BEST PRACTICE: Never use a capital letter for your field names. Use underscores when indicating a space. You can also do a camel or pascal case.

```
user1 = User()
user1.first_name = "John"
user1.last_name = "Doe "

print(user1.first_name)
OUTPUT: John
```

This is an 'instance' of a class or commonly called as an 'Object'

Field: A data attached to an Object

You can access the data or field by putting a dot (.) on the object followed by the field name.

Class scope

- If you create a field name for a specific object, that field name will be only for that object.
- Globally, they won’t exist unless defined or initialized.

Data types for fields

- Having different data types for each field are allowed.

Setting attributes

- If you set an attribute or field for one object while none for the other, only that specific object will be able to access that field.
- If an object user1 and user2 both are initialized with a ‘first_name’. They will both have that field. **user.first_name** and **user2.first_name** will have no errors.
- In case that **user2** had an **age field**, while **user1** had **none**. Putting **user1.age** will result to an **ERROR** indicating that it has **NO ATTRIBUTE ‘age’**.

Class features

- Object Methods
- Initialization/Constructors
- Help text/Doc Strings

What is a constructor?

- Special type of method used to initialize an object.
- Responsible for assigning values to the data members of a class when an object is created.
- Python calls its constructor the __init__ function, short for initialization.

The __init__() Function

- The examples above are classes and objects in their simplest form and are not really useful in real life applications.
- 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:

Python `__init__` Method

The reserved Python method `__init__()` is called the ***constructor*** of a class.

You can call the constructor method to create an object (=instance) from a class and initialize its attributes.

Syntax of constructor declaration

- As we have seen in the above example that a constructor always has a name init and the name init is prefixed and suffixed with a double underscore(__). We declare a constructor using def keyword, just like methods.

```
def __init__(self):
    # body of the constructor
```

Using the __init__ function

Example

Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Constructor Example

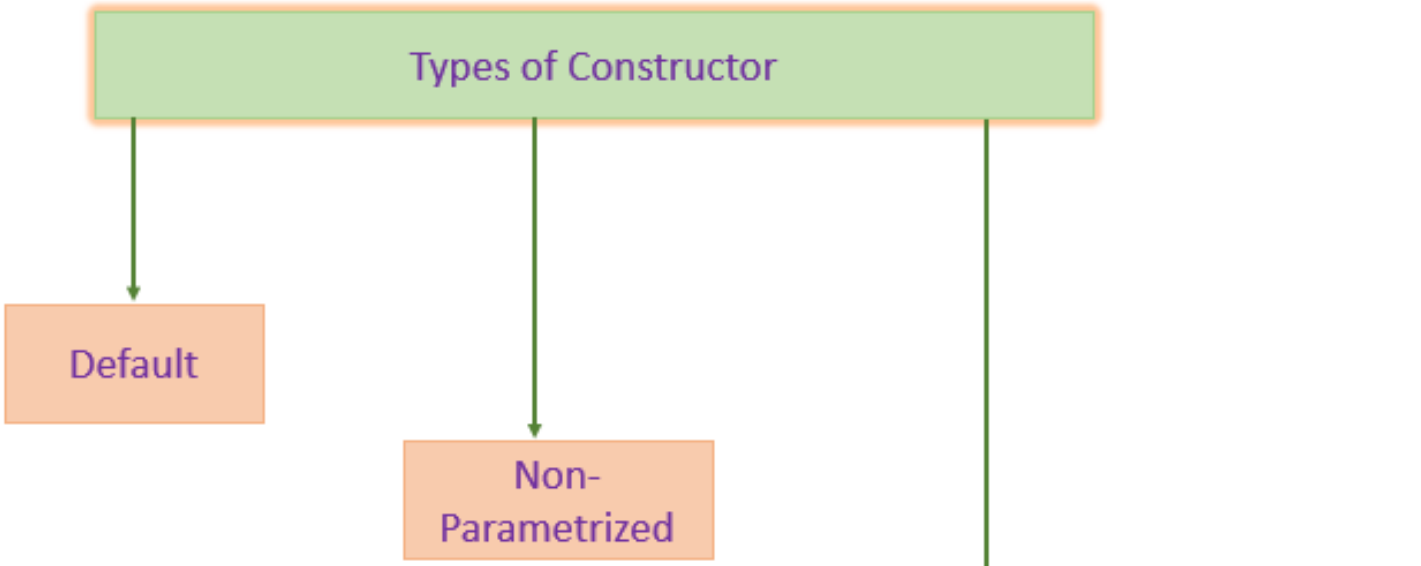
- Here we have an instance variable num which we are initializing in the constructor. The constructor is being invoked when we create the object of the class (obj in the following example).

```
class DemoClass:
    # constructor
    def __init__(self):
        # initializing instance variable
        self.num=100

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class. This invokes constructor
obj = DemoClass()

# calling the instance method using the object obj
obj.read_number()
```

Output:

```
100
```

Types of Constructors



Implicit vs Explicit

- Explicit: When something is explicit, it means it's clearly stated or defined in the code. You directly specify what you want to happen without leaving it to inference.
- Implicit: In contrast, when something is implicit, it means it's implied or understood without being directly stated. The behavior is inferred from the context.
- In Python, **explicitness is generally favored** over implicitness because it leads to more readable and understandable code.

Python – default constructor example

- Note: An object cannot be created if we don’t have a constructor in our program. Therefore, when we do not declare a constructor in our program, python does it for us. Let's have a look at the example below.
- Example: When we do not declare a constructor
- In this example, we do not have a constructor but still we are able to create an object for the class. This is because there is a default constructor implicitly injected by python during program compilation, this is an empty default constructor that looks like this:

```
def __init__(self):
    # no body, does nothing.
```

Class in default without a constructor

```
class DemoClass:
    num = 101

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
obj = DemoClass()

# calling the instance method using the object obj
obj.read_number()
```

Class with a constructor

```
class DemoClass:
    num = 101

    # non-parameterized constructor
    def __init__(self):
        self.num = 999

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
obj = DemoClass()

# calling the instance method using the object obj
obj.read_number()
```

Output:

```
999
```

When to explicitly declare constructors?

- You would declare the __init__ method explicitly when you want to define the initial state of an object when it is created. This allows you to set default values for attributes and perform any necessary initialization steps.
- There may be some cases where you don't need to define an __init__ method explicitly. For example, if your class doesn't have any attributes, or if you don't need to initialize the attributes when the object is created, then you don't need to define an __init__ method.
- **NOTE: In general, you should always define an __init__ method for your class, because it is good practice to initialize the attributes of your objects as soon as they are created.**

Python – Parameterized constructor example

- When we declare a constructor in such a way that it accepts the arguments during object creation then such type of constructors are known as Parameterized constructors.
- Such type of constructors, we can pass the values (data) during object creation, which is used by the constructor to initialize the instance members of that object.

```
class DemoClass:
    num = 101

    # parameterized constructor
    def __init__(self, data):
        self.num = data

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
# this will invoke parameterized constructor
obj = DemoClass(55)

# calling the instance method using the object obj
obj.read_number()

# creating another object of the class
obj2 = DemoClass(66)

# calling the instance method using the object obj
obj2.read_number()
```

Output:

```
55
66
```

The ‘self’ parameter

- A reference to the current instance of the class and is used to access variables that belongs to the class.

- **NOTE: It does not have to be named self , you can call it whatever you like, but it must be the first parameter of any function in the class**

Use the words *mysillyobject* and *abc* instead of *self*:

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

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

More examples to understand ‘self’

- In object-oriented programming, whenever we define methods for a class, we use self as the first parameter in each case.
- Let's look at the definition of a class called Cat.
- In this case all the methods, including __init__, have the first parameter as self.
- Class is a blueprint for the objects. Classes can be used to create multiple numbers of objects. Let's create two different objects from the above class.
- The self keyword is used to represent an instance (object) of the given class. In this case, the two Cat objects cat1 and cat2 have their own name and age attributes. If there was no self argument, the same class couldn't hold the information for both these objects.
- However, since the class is just a blueprint, self allows access to the attributes and methods of each object in python. This allows each object to have its own attributes and methods. Thus, even long before creating these objects, we reference the objects as self while defining the class.

```python
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")
```

```python
cat1 = Cat('Andy', 2)
cat2 = Cat('Phoebe', 3)
```

Why is self explicitly defined everytime?

- Even when we understand the use of self, it may still seem odd, especially to programmers coming from other languages, that self is passed as a parameter explicitly every single time we define a method. As The Zen of Python goes, "Explicit is better than implicit".
- So, why do we need to do this? Let's take a simple example to begin with. We have a Point class which defines a method distance to calculate the distance from the origin.

```
class Point(object):
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y

    def distance(self):
        """Find distance from origin"""
        return (self.x**2 + self.y**2) ** 0.5
```

```
>>> p1 = Point(6,8)
>>> p1.distance()
10.0
```

REMEMBER!

- *self represents the instance of the class.*
- *self is always pointing to the current object.*
- *By using self, we can access the attributes and methods of the class in Python.*
- *self binds the attributes with the given arguments.*
- *Python does not use the "@" syntax to refer to instance attributes.*
- **In summary, self is a reference to the current instance of the class, and it allows you to access the attributes and methods of that instance. By convention, self is the first parameter of any instance method in a class, and it is used to bind the attributes of the instance to the arguments passed to the method.**

Passing but not receiving automatically

- Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.
- EFFECT: distinguishing between instance attributes (and methods) from local variables becomes EASIER!

Objects

- All the objects have a state, behavior and identity.
- **State of an object** - The state or attributes are the built in characteristics or properties of an object. For example, a T.V has the size, color, model etc.
- **Behavior** of the object - The behavior or operations of an object are its predefined functions. For example, a T.V. can show picture , change channels, tune for a channel etc. in object oriented programming terminology the behavior is implemented through methods.
- **Object identity** - Each object is uniquely identifiable. For example, the fridge can not become the T.V.

Another example using the self Parameter

Use the words *mysillyobject* and *abc* instead of *self*:

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

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Object Methods

- Objects can also contain methods.
- Methods in objects are functions that belong to the object.

Modify Object Properties

You can modify properties on objects like this:

Example

Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

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 Person:
  pass
```

Creating a Class with Objects

Insert a function that prints a greeting, and execute it on the p1 object:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Class

Constructor

***'self'* is indicating that the attributes are from itself**

***'self'* is indicating that the method is from itself.**

Object Method/Function

Object

Object calling its OWN method

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.

Doc string

- Provides a more detailed information about the class.
- The doc string can be added into the class by using a triple quote written after the first line of the class.

```
class User:
  """A member of FriendFace.  For now we are
    only storing their name and birthday.
    But soon we will store an uncomfortable
    amount of user information."""
```

Class →

Object

Pokemon
Name: Pikachu
Type: Electric
Health: 70
attack()
dodge()
evolve()

Attributes

Methods

Can you make classes, objects, and methods of us?


Next meeting, Inheritance