Digital Career Institute

Python Course - OOP Concepts

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to introduce the students to Object Oriented Programming concepts.

By the end of this submodule, you should be able to understand:

- the difference between a class and an object instance,
- how classes are implemented in Python,
- the concept of inheritance, and
- the convention on constants, private and protected attributes and methods in Python.

Topics

- Introduction to OOP
- Classes & Objects
  - Classes
    - Fields
    - Methods
    - Constructors
  - Objects
  - Difference between object and classes
- Inheritance
  - Inheritance types
  - ***super()*** built-in function
  - self
  - multiple inheritance and **super()**
- Visibility of attributes/methods:
  - ***public*** (default)
  - ***private***
  - ***protected***
  - ***constants***

Digital Career Institute

Glossary

Term	Definition
OOP	Object Oriented Programming - refers to the programming paradigm
Class	A template/blueprint that describes the behavior/state that the object of its type support
Object	A concrete instance of a specific Class
Inheritance	The process of inheriting (extending) the behaviors/states of another class
IS-A	A term used to define the inheritance relationship, where X IS-A Y if X inherits from Y, by extending (Y is a class) or implementing (Y is an interface)

Introduction to OOP

Digital Career Institute

Introduction to OOP

Object Oriented programming (OOP) is a programming paradigm that relies on the concept of **classes** and **objects**. It is used to structure a software program into simple, reusable pieces of code blueprints (classes), which are used to create individual instances (objects). There are many object-oriented programming languages including **Java**, **JavaScript**, **C++** and **Python**.

A **class** is an abstract blueprint used to create more specific, concrete objects. Classes often represent broad categories, like Car or Dog that share attributes. These classes define what attributes an instance of this type will have, like color, but not the value of those attributes for a specific object.

Class templates are used as a blueprint to create individual **objects**. These represent specific examples of the abstract class, like ***my_car*** or ***golden_retriever***. Each object can have unique values to the properties defined in the class.

Introduction to OOP - Classes & Objects

Digital Career Institute

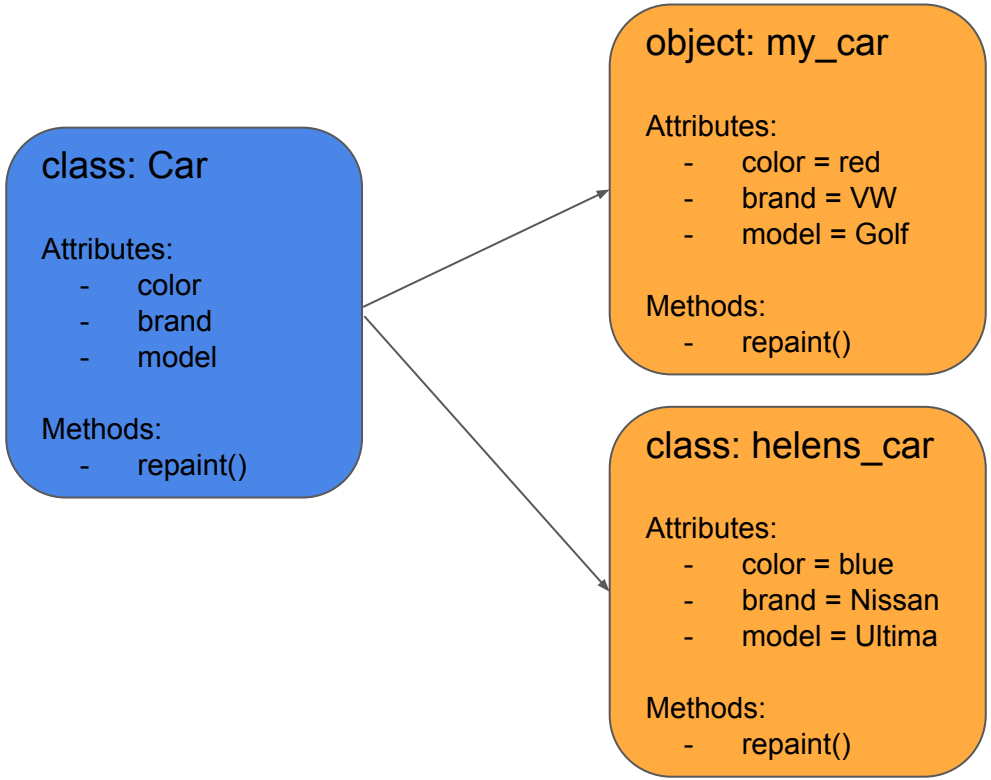
If we have a **class *Car***, then it should contain all the properties a car must have: *color*, *brand*, and *model*. We then create an instance of a Car type **object**, *my_car*, to represent my specific car.

Set the value of the properties defined in the class to describe my car, without affecting other objects or the class template.

Reuse this class to represent any number of cars.

Our Car class may also have a method called *repaint* that changes the *color* attribute of our car. This function is only helpful to objects of type ***Car***, so we declare it within the Car class thus making it a method.

Introduction to OOP - Classes & Objects



Class blueprint being used to create two Car type objects, my_car and helens_car

Introduction to OOP

Digital Career Institute

Why is OOP important?

- **Reduces complexity:** OOP promotes the **reuse of data**, helping **reduce development time and complexity**. Using OOP concepts in Python, you can **write a functionality once and reuse it everywhere else**.

- **Reduces maintenance time:** OOP makes projects modular, allowing you to **isolate and solve issues easier**. For example, if the bill amount is not right, it means that the problem is with the Bill class and one can go directly there and start debugging.

- **Widely applicable:** OOP can be used to **model any scenario imaginable**, making it highly useful and applicable in a variety of business use cases.

Introduction to OOP - Classes & Objects

Illustrates class and object relationship through cookie-cutter and cookies.

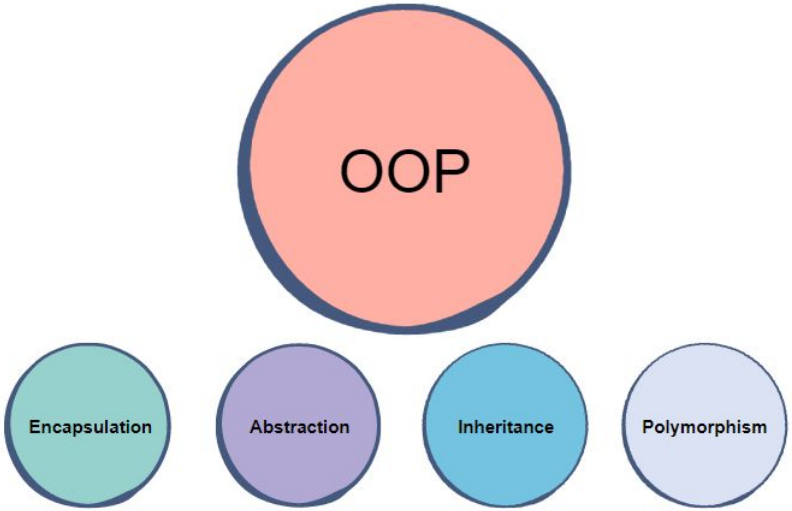
Introduction to OOP - Principles

Digital Career Institute

Four Principles of OOP

The four pillars of object oriented programming are:

- **Inheritance**: child classes inherit data and behaviors from parent class
- **Encapsulation**: containing information in an object, marking only selected information as being public. This is only partially possible in Python.
- **Abstraction**: only exposing high level public methods for accessing an object
- **Polymorphism**: many methods can do the same task



At the core of the lesson

Digital Career Institute

Benefits of OOP

- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them

Classes & Objects

Classes & Objects - Classes

Digital Career Institute

Classes

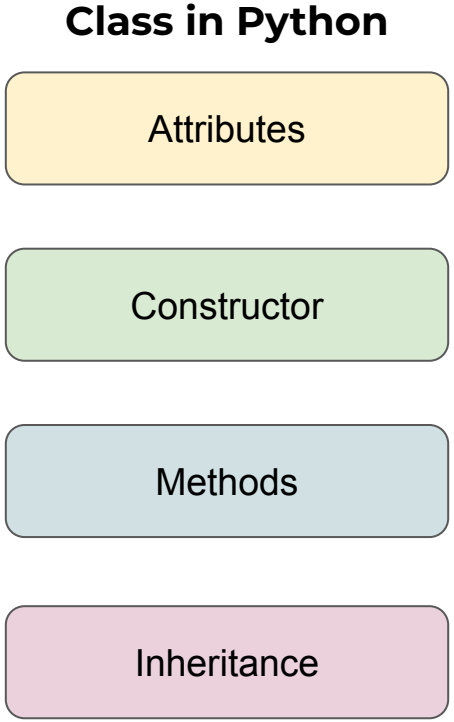
A class is a **template** (or **blueprint**) from which objects are created.

Classes define **states** as **instance variables** and **behaviors** as **instance methods**.

In a nutshell, classes are essentially **user defined data types**.

Classes are where we create a blueprint for the structure of methods and attributes.
Individual objects are instantiated, or created from this blueprint.

Classes & Objects - Classes



Python classes and their subcomponents

Classes & Objects - Classes

Digital Career Institute

Attributes

- Attributes are the information that is stored.
- Attributes are ideally defined in the **__init__()**-method or set on the class itself as part of the class definition .
- When objects are instantiated individual objects contain data stored in the attributes.
- The state of an object is defined by the data in the object's attributes.
- For example, a puppy and a dog might be treated differently at pet camp. The birthday could define the state of an object, and allow the software to handle dogs of different ages differently.

Classes & Objects - Classes

Digital Career Institute

Methods

- Methods represent **behaviors**.
- Methods perform **actions**.
- Methods might return **information** about an object, or **update** an object's data.
- The method's code is defined in the class definition.
- When individual objects are instantiated, these objects can call the methods defined in the class.
- Methods often modify, update or delete data. They don't have to update data, though.
- Methods are how programmers promote **reusability**, and keep **functionality encapsulated** inside an object. This reusability is a great benefit when **debugging**. If there's an error, there's only one place to find it and fix it instead of many.

Classes & Objects - Classes

Digital Career Institute

The constructor method

- The **__init__()**-constructor is a **special method** in Python.
- It is called when an instance of object is created.
- It is called constructor because it constructs the values at the time of object creation.
- It is not necessary to write a constructor for a class. If there is no **__init__()**-method, the Python compiler will treat it as if there was an empty **__init__()**-method.
- Each time an object is created, the **__init__()**-method is invoked to assign initial values to the fields of the class.

Classes & Objects - Objects

Digital Career Institute

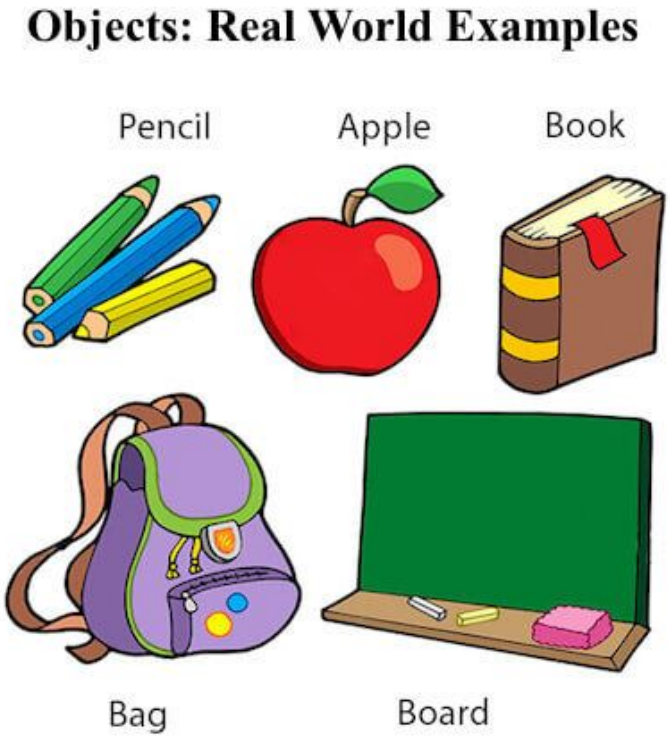
Objects

Objects are **instances** of classes created with **specific data**. It's an **entity** that has **states** and **behaviors**.

An object is also referred to an **instance** of a class. Instantiating a class means the same thing as **creating an object**.

Software objects are the actual **representation** of real world objects.

Classes & Objects - Objects



Source: JavaTPoint

Classes & Objects - Objects

Digital Career Institute

Characteristics of Objects

State
Represents the data of an object stored in its attributes.

Behavior
Represents the way the object behaves when its methods are invoked.

Identity
The Python interpreter needs to identify each object uniquely.

Classes & Objects - Difference

Digital Career Institute

	What is it?	Information Contained	Actions	Example
Classes	Blueprint	Attributes	Behaviors defined through methods	class Car
Objects	Instance	State, Data	Methods	my_car, helens_car

At the core of the lesson

Digital Career Institute

What are Classes?

- A class is a template from which an object can be instantiated from.
- Classes define states as **instance variables** and behaviors as **instance methods**.
- Instance variables are also known as **member variables** or **fields**.
- Classes don't consume any space.
- In Python, every class extends from a built-in class called **object**.

What are Objects?

- An object is an instance of a class, with its own states and behaviors.
- The **attributes** of an object defines its state.
- The methods of an object defines its behaviors.
- In Python, every object instance is also an instance of the built-in class called **object**.

Inheritance

Inheritance

Inheritance

Inheritance allows classes to **inherit** features of other classes.

Parent classes extend attributes and behaviors to child classes.

If basic attributes and behaviors are defined in a parent class, child classes can be created extending the functionality of the parent class, and adding additional attributes and behaviors.

The benefits of inheritance are that programs can create a generic parent class, and then create more specific child classes as needed.

This simplifies overall programming, because instead of recreating the structure of the generic class multiple times, child classes automatically gain access to functionalities within their parent class.

Inheritance uses a parent-child relationship (IS-A relationship).

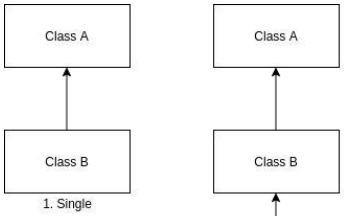
Inheritance - What is Inherited

A **class** can **extend one or multiple classes**. A class can also **extend a class that already extends another class**, thus creating a multilevel inheritance.

What is inherited?

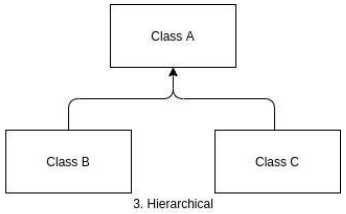
1. Methods.
2. Attributes.

Inheritance - Types

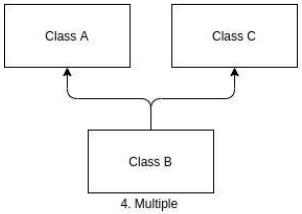


1. Single

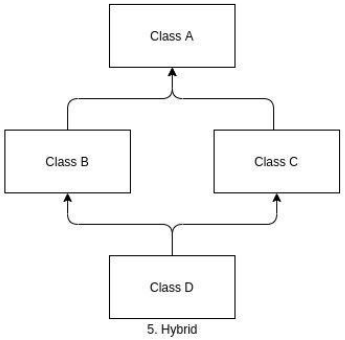
2. Multi level



3. Hierarchical



4. Multiple



5. Hybrid

1. **Single**: When a *class* ***extends*** *from another class* (either concrete or abstract).
2. **Multi level**: When a *class* ***extends*** *from another class that already extends another class*.
3. **Hierarchical**: When two or more different classes **extend** from another class.
4. **Multiple**: *W*hen a class **extends** multiple interfaces.
5. **Hybrid**: *A* mix of *multi level, hierarchical and multiple* inheritance types.

Inheritance - super()

Digital Career Institute

The *super()* built-in function

The **`super()`** built-in function in Python plays a major role in Inheritance because we can use it to **refer to parent class methods and fields**.

In other words, we can use the **`super()`** built-in function to call methods and data members of the immediate parent class.

- Whenever we create an instance of the child class, then **the instance of the parent class is created implicitly**. A reference can be obtained by calling the **super()** built-in function.
- If we have the **same method name in a child as well as parent** class then the **`super()`** built-in function is used to call the parent class method.
- **`super().__init__(...)`** is used to invoke the super-class's **constructors**

Inheritance - super()

```
>>> class Animal:
...   def __init__(self, name):
...     self.name = name
...   def display(self):
...     print(f"Name: {self.name}")
...
>>> class Fish(Animal):
...   def display(self):
...     print("Type: Fish")
...     super().display()
...
```

```
>>> my_fish = Fish('Trout')
>>> my_fish.display()
Type: Fish
Name: Trout
```

Invoking the display() method of the parent class with the super() built-in function

Inheritance - super()

```
>>> class Animal:
...   def __init__(self, name):
...     self.name = name
...     print(f"Name: {self.name}")
...
>>> class Bird(Animal):
...   def __init__(self, name, flying):
...     super().__init__(name)
...     self.can_fly = can_fly
...     print(f"Flying: {self.can_fly}")
...
```

```
>>> my_bird = Bird('Penguin', False)
Name: Penguin
Flying: False
```

Calling the constructor of the parent class with the super() built-in function.

Inheritance - self

self

As the name defines, **`self` refers to the current object** and it is a **reference variable**. It is used to **refer to the current object inside a method or a constructor**.

`self` is used in various contexts as given below:

- To refer to the **instance variables and methods of current class**
- Is automatically passed as the first **argument in all method calls**
- Can be used to **return the current class instance**
- It's mostly used for **ambiguity in variable names inside the same scope**

Note: In Python, the use of *self* is merely a convention. It is not a keyword. You can use other terms instead. The term that is used within a method is the name of the first argument of that method.

Inheritance - self

```
>>> class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
...     def show(self):
...         print(f"{self.name}
{self.age}")
...
...     def info(self):
...         print(self)
```

```
>>> me = Person('Thomas', 35)
>>> me.show()
Thomas 35
>>> me.info()
<__main__.Person at 0x7f9598cb2f70>
>>> print(me)
<__main__.Person at 0x7f9598cb2f70>
```

self is a reference to the object itself.

Inheritance - super() & self

	`super()`	`self`
Definition	Refers to the immediate parent class instance	Refers to the current class instance
Invoke	Can be used to invoke immediate parent class method	Can be used to invoke current class method
Constructor	`super().__init__()` references the constructor of the immediate parent class	`self.__init__()` references the constructor of the current class
Override	When invoking a superclass version of an overridden method the `super()` `built-in function` should be used	When invoking a current version of an overridden method the `self` keyword can used

Inheritance - multiple and super()

Digital Career Institute

super() and single inheritance

In very simple cases, one can reference the parent class directly and achieve the same result:

```
class Higher:
    def __init__(self):
        print('Higher')

class Lower(Higher):
    def __init__(self):
        print('Lower')
        super().__init__()

>>> a = Lower()
Lower
Higher
```

```
class Higher:
    def __init__(self):
        print('Higher')

class Lower(Higher):
    def __init__(self):
        print('Lower')
        Higher.__init__(sel
f)

>>> a = Lower()
Lower
Higher
```

Inheritance - multiple and super()

```
class Top:
      def __init__(self):
            print('Top')

class Middle1(Top):
      def __init__(self):
            print('Middle1')
            Top.__init__(self)

class Middle2(Top):
      def __init__(self):
            print('Middle2')
            Top.__init__(self)

class Bottom(Middle1, Middle2):
      def __init__(self):
            print('Bottom')
            Middle1.__init__(self)
            Middle2.__init__(self)
```

multiple inheritance without super()

```
>>> a = Bottom()
Bottom
Middle1
Top
Middle2
Top
```

Note: Top is initialized twice

Inheritance - multiple and super()

Digital Career Institute

```
class Top:
      def __init__(self):
            print('Top')

class Middle1(Top):
      def __init__(self):
            print('Middle1')
            super().__init__()

class Middle2(Top):
      def __init__(self):
            print('Middle2')
            super().__init__()

class Bottom(Middle1, Middle2):
      def __init__(self):
            print('Bottom')
            super().__init__()
```

multiple inheritance with super()

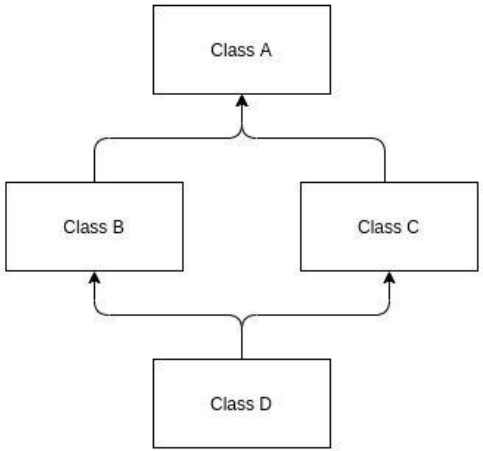
```
>>> a = Bottom()
Bottom
Middle1
Middle2
Top
```

Note: the usage of super() ensures that Top is only initialized once.

Inheritance - multiple and super()

The diamond-shape problem

- If one goes up in the hierarchy of classes, eventually all classes will derive from object.
- Therefore, whenever using multiple inheritance, all the parents will have the same ancestors if one goes up the hierarchy sufficiently high.
- In order to prevent calling the same method on an ancestor several time, one can use the **super()** built-in function.



At the core of the lesson

Digital Career Institute

Inheritance

- Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).
- In Python, inheritance is performed by using the parent class as a parameter when creating the child class.
- In Python, inheritance is an is-a relationship. That is, we use inheritance only if there exists an is-a relationship between two classes.
- Python allows multiple inheritance and the super() built-in methods allows us to avoid the diamond-shape problem.

Encapsulation

Visibility - private and protected

Different from languages such as **Java** or **JavaScript**, in **Python**, all attributes and methods are public and they can be overridden in child classes. We therefore instead rely on naming conventions to mark which variable or method is to be treated as if it were private and which variables are to be treated as constants.

- **Default attributes/methods:** Name is in lowercase snake case.
- **Private attributes/methods:** Name is in lowercase snake case and prefixed with one underscore.
- **Protected attributes/methods:** Name is in lowercase snake case and prefixed with two underscores.
- **Also - constant attributes:** Name is in uppercase snake case.

Note: Following the naming convention for protected attributes/methods will trigger name mangling. They will still be accessible, but not under the name originally assigned.

Encapsulation - private and protected

Digital Career Institute

```
from datetime import datetime

BOILINGPOINT_OF_WATER = 100

class MyClass:
      def __init__(self, first_name):
            self.first_name = first_name
            self._creation_date = datetime.now()
            self.__internal_counter = 0

>>> a = MyClass('John')
>>> a.name
John
>>> a._creation_date
datetime.datetime(2021, 7, 23, 1, 58, 40, 160677)
>>> a.__internal_counter
AttributeError: 'MyClass' object has no attribute '__internal_counter'
>>> a._MyClass__internal_counter
0
```

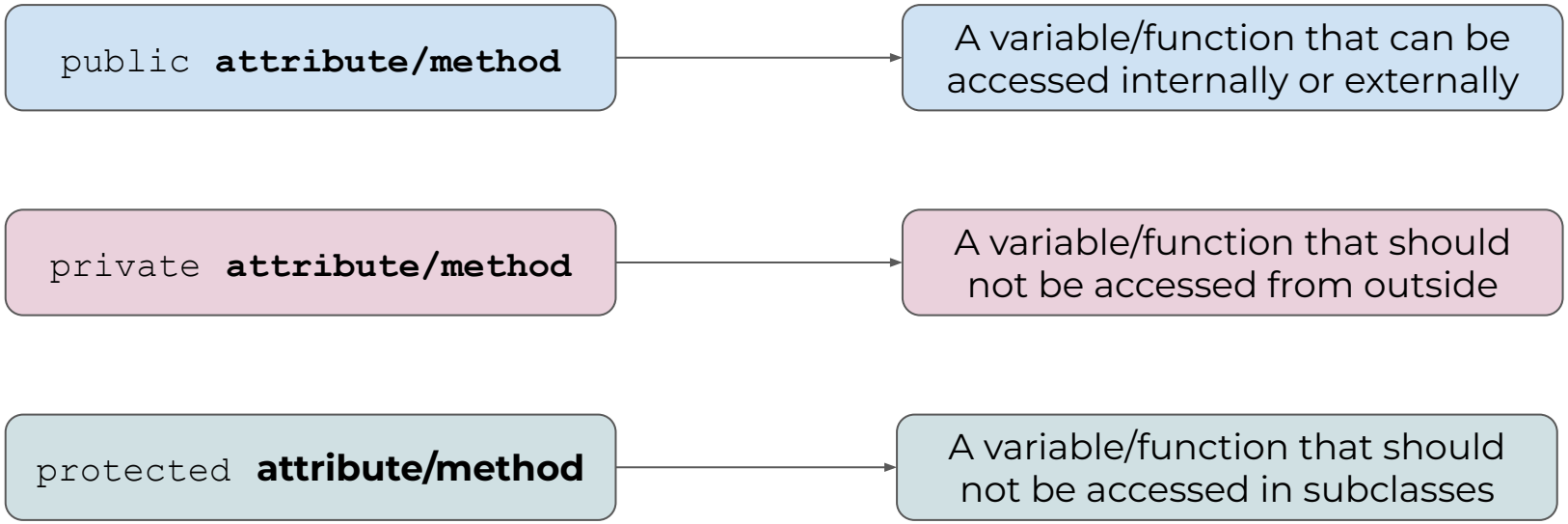
Constant: snake case, capital letters, usually set outside of class

Default: snake case, lowercase letters

Private: underscore + snake case, lowercase letters

Protected: 2 x underscore + snake case, lowercase letters

Encapsulation



Note: In Python, the methods and variables of a class/object are all public. Naming convention tells developers how they should restrain themselves, but it does not actually prevent them from doing something else.

At the core of the lesson

Digital Career Institute

Encapsulation

- Python does not have the same capabilities that other languages have to protect and hide attributes and methods from descending classes.
- Python relies on naming conventions to communicate to other developers how a particular attribute/method should be treated.

Reflection Round

Digital Career Institute

Documentation

Documentation

Digital Career Institute

1. The Python tutorial on classes (docs.python.org)
2. Object-Oriented Programming (OOP) in Python 3 (realpython.com)
3. Java Classes and Objects (w3schools.com)
4. Python Object Oriented Programming (programiz.com)
5. self in Python class (geeksforgeeks.org)
6. Python super() (programiz.com)
7. Python’s super() considered super! (rhettinger.wordpress.com)
8. Function and Variables Names to Constants (python.org)

Digital Career Institute