**Classes**

Python’s core data structure:

* Lists
* Tuples
* Dictionaries
* Etc.
* **Classes** are part of a programming paradigm called **object-oriented programming (OOP).**
* **OOP** focuses on building **reusable blocks of code** called **classes**.
* When you want to **use a class in one of your programs**, you make an **object from that class**, which is where the phrase "object-oriented" comes from.
* A **class** is a body of code that **defines the attributes and behaviors** required to accurately model something you need for your program. You can model something from the real world, such as a rocket ship or a guitar string, or you can model something from a virtual world such as a rocket in a game, or a set of physical laws for a game engine.
* An **attribute** is a **piece of information**. In code, an attribute is just **a variable that is part of a class**.
* A **behavior** is an **action that is defined within a class**. These are **made up of methods**, which are just **functions** that are **defined for the class.**
* An **object is a particular instance of a class**. An object has a certain set of values for all of the attributes (variables) in the class. You can have as many objects as you want for any one class.
* **Class**
* **Object**
* **Attribute**
* **Behavior**
* **Method**

Once you have a **class**, you can define an **object** and use its **methods**.

**1. Class:**

Names of classes are following the PascalCase formatting convention.

More on comments in Classes [here](https://peps.python.org/pep-0257/#handling-docstring-indentation).

A screenshot of a computer program

Description automatically generated with low confidence

**2. Object** (created from the Class):Using this example: it creates a rocket (a variable called my\_rocket). An object *Rocket* is being created from the class.



**1.2 Method 1 & 2** (within the Class):

A method is a function that is part of a class. It sets the values for any parameters that need to be defined when an **object** is first created.

Function names that start and end with two underscores are special built-in functions. The \_\_init\_\_() is called automatically when you create an object from your class.

All methods in a class need the self object as their first argument, so they can access any attribute that is part of the class.

In this case the method 1 initializes the x and y values of the Rocket to 0.

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with low confidence

* Access **object’s** **variables** or **methods** by using **object’s name** and **value name after self.**: my\_rocket.x or my\_rocket.y
* Use a **method** on an **object** you write **object’s name** and **method’s name**: my\_rocket.move\_up()

**Example 1**

A screenshot of a computer program

Description automatically generated with low confidence

Setting two objects (in this example *rockets*), moving them, and then getting the distance between them:

A screenshot of a computer code

Description automatically generated with low confidence

**Example 2**

A screenshot of a computer program

Description automatically generated with medium confidence

**Example 3**

A screenshot of a computer program

Description automatically generated with medium confidence

**Use help() to get the notes / documentation of a class or a function:**

A screenshot of a computer program

Description automatically generated with medium confidence

**Inheritance of classes**

One class can inherit from another class. This means you can **base a new class on an existing class**; the **new class *inherits* all of the attributes and behavior of the class it is based on**.

A new class can override any undesirable attributes or behavior of the class it inherits from, and it can add any new attributes or behavior that are appropriate.

The original class is called the **parent class** or **superclass**, and the new class is a **child of the parent class** or a **subclass**.

Using **Example 2 as parent class** here is an **example of a child class**:

Parent class should be in the parentheses; in the first line you mention the parents’ class variables and new variables (*university, graduated, gpa* in this example); in the second line after super() mention the original variables (see note below); after this line mention the new variables.

A screenshot of a computer program

Description automatically generated with low confidence

Define a student object:

A screenshot of a computer program

Description automatically generated with low confidence

Methods from the original class work fine:

A screenshot of a computer program

Description automatically generated with low confidence

Added values from the child class work fine as well:

A screenshot of a computer program

Description automatically generated with low confidence

**Note on** super()**:** These lines are the same, but the first code is better because it excludes the name of the parent class and that makes it easier to work without the need to update the name of the class:

.super().\_\_init\_\_(name, age, birthplace, city, function)

or

Person.\_\_init\_\_(self, name, age, birthplace, city, function)

**Modules**

**Saving classes in modules**

When you save a class into a separate file, that file is called a **module**. You can have any number of classes in a single module. There are a number of ways you can then import the class you are interested in.

The module is saved with a lowercase name, e.g. file name is: *rocket.py*.

**Import class from a module. Option 1**

from module import Class1, Class2

**Import classes** *Rocket* (parent) and *Shuttle* (child) **from a module** called *rocket.py*:



The problem with this method is that names of classes that you are importing can conflict with names that have already been used in the program you are working on. In this case you can use alternative options described below.

**Option 2**

import module

Class1NewName = module.Class1()

Class2NewName = module.Class2()

From module *rocket.py* import *\*new name for class A\** = *module name.class A*

A picture containing text, font, white, screenshot

Description automatically generated

Note that in this method class Rocket should be renamed to rocket\_0 because the module has the same name rocket.

**Option 3**

import module as module\_new\_name

Class1Name = module\_new\_name.Class1()

Class2Name = module\_new\_name.Class2()

When importing a module you are free to choose any name you want for the module in your project. This is used to shorten the name of the module, so you don't have to type a long module name before each class. Note that it’s better to give a meaningful new name like rocket\_module instead of just r, so that others reading your code get a better grip of your code.

A picture containing text, font, screenshot, white

Description automatically generated

**Option 4**

from module import \*

**This is not recommended**, for a couple reasons. First of all, you may have no idea what all the names of the classes and functions in a module are. If you accidentally give one of your variables the same name as a name from the module, you will have naming conflicts. Also, you may be importing way more code into your program than you need.

**Saving functions in modules**

You can use modules to store a set of functions that you want to use in different programs. These functions do not have to be attached to a class.

Importing a function from a module follows the same rules as importing a class from a module. In the options below multiplying is a *module* and double, triple, and quadruple are *functions* in that *module*.

**Option 1**

from module import Function1, Function2

**A picture containing text, font, screenshot, white

Description automatically generated**

**Option 2**

import module

Function1NewName = module.Function1()

Function2NewName = module.Function2()

**A picture containing text, font, screenshot, white

Description automatically generated**

**Option 3**

import module as module\_new\_name

Function1Name = module\_new\_name.Function1()

Function2Name = module\_new\_name.Function2()

**A picture containing text, font, screenshot, white

Description automatically generated**

**Option 4**

from module import \*

**A screenshot of a computer code

Description automatically generated with low confidence**

**Name mangling**

Name mangling is a feature specific to classes in Python. It is used to localize names within a class to avoid unintended name conflicts with subclasses or other classes. The purpose of name mangling is to provide a way to indicate that certain attributes or methods are intended for internal use within the class and should not be accessed or overridden outside of the class.

When a name is mangled, it is prefixed with a double underscore \_\_ and a prefix based on the class name. This ensures that the name becomes unique within the class hierarchy. However, name mangling does not provide true data encapsulation or privacy, as the mangled names can still be accessed from outside the class using the mangled syntax \_ClassName\_\_mangled\_name (example below):

A screenshot of a computer program

Description automatically generated with low confidence

Without \_Length this code will return an error:

A picture containing text, screenshot, font, line

Description automatically generated

Therefore, name mangling is most commonly used within classes to indicate the intended scope and visibility of attributes and methods. It helps to prevent accidental name clashes and encourages encapsulation by signaling that certain names are intended for internal use only.

Using the above example:

If you write \_\_l instead of l in the \_\_add\_\_ method, it will simply be treated as a different variable name (like ‘name’ instead of ‘l’). The use of double underscores (\_\_) at the beginning of a variable name **within a method** does not trigger name mangling or have any special behavior. It will be considered as a regular variable name and not affect the functionality of the code.

Variables with double underscores at the beginning (\_\_variable) trigger name mangling when they are class attributes. This behavior is not applicable to local variables within methods or functions.

**@property**

When you use @property before a method in a class, it **transforms the method into a read-only property**. It allows you to access the method like an attribute, without using parentheses to invoke it as a function.

For example we have two classes:

Class 1 with @property: Class 2 without @property:

A screen shot of a computer code

Description automatically generated with low confidence A screenshot of a computer code

Description automatically generated with low confidence

When you assign and run values you get this:

A picture containing text, screenshot, font, white

Description automatically generated A screenshot of a computer

Description automatically generated with low confidence

Or simply

A picture containing text, screenshot, font, white

Description automatically generated

In the first case, since *area* is decorated with @property, you can access it directly as an attribute (my\_circle.area) instead of calling it as a method (my\_circle.area()). This makes the code cleaner and more intuitive.

A screenshot of a computer

Description automatically generated with medium confidence A picture containing text, screenshot, font, white

Description automatically generated

However, because it is a read-only property, you cannot assign a new value to it. If you try to assign a new value like my\_circle.area = 100, it will raise an ***AttributeError*** because the property is read-only.

So, by using @property, you can achieve a more convenient and natural way of accessing calculated or derived values as if they were regular attributes, but you cannot assign new values to them directly.