[Import modules 2](#_Toc104406274)

[List of objects 3](#_Toc104406275)

[Composition 4](#_Toc104406276)

[Composition or inheritance? 5](#_Toc104406277)

[The \_\_str\_\_ method 6](#_Toc104406278)

# Import modules

To better organise the code, one can write code in separate files /modules and import the module to the main file. For example, a class definition can be written in a separate file and imported to the main file to keep the code cleaner. It could potentially cause many files to one project.

|  |
| --- |
| **class** **Employee**:  **def** **\_\_init\_\_**(self, name, title):  self.name = name  self.title = title    **def** **display**(self):  print(f'Employee: {self.name}')  print(f'Title: {self.title}') |

but the employee can be instantiated in another file, by importing the class from the former file. To instantiate the Employee class, the code would look like this:

|  |
| --- |
| *# This file is the program*  from class\_definition import Employee  emp = class\_definition.Employee("Marcia", "VP of Sales")  emp.display() |

Functions can also be imported. To continue the example code above, a function to add would be a greeting.

|  |
| --- |
| **def** **greeting**():  print('Hello from the "class\_definition" module') |

To import a function in the program file, the code would look like this:

|  |
| --- |
| *# This file is the program*  **from** class\_definition **import** Employee, greeting  emp = Employee("Marcia", "VP of Sales")  emp.display()  greeting() |

# List of objects

To instantiate several objects of a class, a list can be created to organise the code. In addition to simplify the code, one can iterate through the list rather than manipulate each object separately.

|  |
| --- |
| *# define the App class*  **class** **App**:  **def** **\_\_init\_\_**(self, name, description, category):  self.name = name  self.description = description  self.category = category    **def** **display**(self):  print(f'{self.name} is a(n) {self.category} app that is {self.description}.') |

To import objects from a CSV file, objects can be imported rather than individually created. To instantiate the class from a CSV file, the code would look like this:

|  |
| --- |
| *# list of objects*  **from** csv **import** reader  **from** app **import** App  apps = []  **with** open('code/advanced/apps.csv') **as** csv\_file:  csv\_reader = reader(csv\_file, delimiter=',')  next(csv\_reader)  **for** name, description, category **in** csv\_reader:  apps.append(App(name, description, category))    print(apps) |

It returns a list of objects that, which may look like <app.App object at 0x7f6b5af70d30> as that is how Python represents an object. The string of numbers is the location of the object.

To interact with the objects, I can iterate through the list of objects.

The code would look like this:

|  |
| --- |
| *# list of objects*  **from** csv **import** reader  **from** app **import** App  apps = []  **with** open('code/advanced/apps.csv') **as** csv\_file:  csv\_reader = reader(csv\_file, delimiter=',')  next(csv\_reader)  **for** name, description, category **in** csv\_reader:  apps.append(App(name, description, category))    **for** app **in** apps:  app.display() |

# Composition

Composition in code would be represented like the code below. To have the engience as its own class makes the code cleaner and easier to read, and it looks less messy. Diagram

Description automatically generated

|  |
| --- |
| class Car:  def \_\_init\_\_(self, make, model, year, engine):  *self*.make = make  *self*.model = model  *self*.year = year  *self*.engine = engine    def describe(self):  print(f'{*self*.year} {*self*.make} {*self*.model}')  class Engine:  def \_\_init\_\_(self, configuration, displacement, horsepower, torque):  *self*.configuration = configuration  *self*.displacement = displacement  *self*.horsepower = horsepower  *self*.torque = torque    def ignite(self):  print('Vroom, vroom!')  my\_engine = Engine("V8", 5.8, 326, 344)  my\_car = Car("De Tomaso", "Pantera", 1979, my\_engine)  my\_car.engine.ignite() |

## Composition or inheritance?

you have the Vehicle class and you want to make a Car class. Ask yourself if a car has a vehicle or if a car is a vehicle. A car is a vehicle; therefore you should use inheritance. Now imagine that you have a Phone class and you want to represent an app for the phone. Ask yourself if a phone is an app or if a phone has an app. A phone has an app; therefore you should use composition.

# The \_\_str\_\_ method

Python returns the location of an object in the memory, therefore the \_\_str\_\_ method is used to return an object in a way that would make sense to us.

For instance, to instantiate an object have it returned on the screen with its parameters as a string instead of its location, the \_\_str\_\_ method is used. In code it looks like this:

|  |
| --- |
| **class** **Dog**:  **def** **\_\_init\_\_**(self, name, breed):  self.name = name  self.breed = breed    **def** **\_\_str\_\_**(self):  **return** f'{self.name} is a {self.breed}'    my\_dog = Dog('Rocky', 'Pomeranian')  print(my\_dog) |