# CS: Objects in Python

## Advanced Topics

## Advanced Topics

## Learning Objectives - Advanced Topics

---

- **Set up and import an object defined in a separate file**
- **Create and manipulate a list of objects**
- **Define object composition**
- **Compare and contrast component and composite classes**
- **Identify when to use composition and when to use inheritance**
- **Compare and contrast the `__repr__` and `__str__` methods**

# Importing Modules

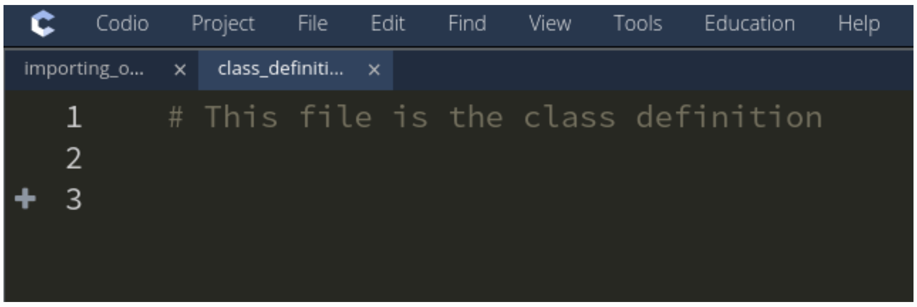
## Importing Module

You may have noticed that writing your own objects adds many lines of code before your the logic of your program even begins. To better organize your code, define classes in a separate module. A module is a file that contains Python definitions or executable statements. Then import the module into your program so you can use the class.

Make sure you are in the file for defining the class (look at the comments at the top of the files). Start by creating a simple `Employee` class.

### ▼ Switching Between Files

Notice that the IDE on the left now has more than one file. Click on the tabs on the top to switch between the files.



```
# This file is the class definition
```

Switching Files

```
# This file is the class definition

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}')
```

Now go to the file for your program (look at the comments at the top of the files). Use the `import` command followed by the name of the Python module, `class_definition`. The name of the module is the same as the name of the Python file minus the `.py` extension. In the example below, the Python file is `class_definition.py`. Therefore the name of this module is `class_definition`.

We want to instantiate an `Employee` object. However, this class is in a different module. You need to tell Python where to find the `Employee` class, so start with the module name `class_definition` followed by a `.`, then instantiate the object.

```python
# This file is the program

import class_definition

emp = class_definition.Employee("Marcia", "VP of Sales")
emp.display()
```

challenge

## Try this variation:

Writing `class_definition.Employee` is a bit long. Python is all about simplicity, so Python provides another way of importing modules in a more explicit manner. Instead of importing the entire module, tell Python to import a specific class from a specific module. Now you no longer have to write `class_definition.Employee`. You can use `Employee` as you normally would.

```python
# This file is the program

from class_definition import Employee

emp = Employee("Marcia", "VP of Sales")
emp.display()
```

## Importing Functions

A Python module can also contain function definitions, which can be imported just like a class. Make sure that you are in the `class_definition` module and then add the function `greeting`.

```python
# This file is the class definition

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}')

def greeting():
  print('Hello from the "class_definition" module')
```

Switch over to your program. Because the `Employee` class and the `greeting` function are located in the same module, we can import each item with a single `import` statement. Be sure to use a comma to separate each item to import.

▼ **Importing Everything From a Module**

If a module has several things that you want to use, Python allows you to import the entire module all at once.

```python
from class_definition import *
```

The `*` is a wildcard that means everything. However, the official Python style guide recommends that you avoid the wildcard and import item separately.

```python
# This file is the program

from class_definition import Employee, greeting

emp = Employee("Marcia", "VP of Sales")
emp.display()

greeting()
```

# List of Objects

## List of Objects

You may find yourself needing several instances of a class. Keeping these objects in a list is a good way to organize your code. It also simplifies interacting with the objects because you can iterate through the list as opposed to manipulating each object separately.

The first thing you need to do is to create a class. We are going to make a class to represent the apps on your smartphone.

```
# 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.desc
```

Next, we need to create a list with objects of the `App` class as each element. To speed this up, we are going to read from a CSV file that has the information for the `name`, `description`, and `category` attributes.

▼ **Why use a CSV file?**

This page is about manipulating a list of objects. Instead of manually creating several objects, we are going to read information from the `apps.csv` file and use it to create several objects in a simple loop.

```
  name,description,category
  Gmail,the official app for Google's email service,communication
  FeedWrangler,used to read websites with an RSS feed,internet
  Apollo,used to read Reddit,social media
  Instagram,the offical app for Facebook's Instagram service,social
media
  Overcast,used to manage and listen to podcasts,audio
  Slack,the official app for Slack's email
replacement,communication
  YouTube,the official app for Google's video service,video
  FireFox,used to browse the web,internet
  OverDrive,used to checkout ebooks from the library,ebooks
  Authenticator,used for two-factor authentication,internet
```

Make sure that you are altering the `list_of_objects.py` file. We need to import the `csv` module and the `App` class from the module `app`. Then create an empty list that will be used to hold each `App` object. Set up a CSV reader to read the file. The first row is composed of the header information, so skip it with the `next function`. Remember, the CSV reader reads each line of the CSV file as a list of strings. Use unpacking to refer to each element as `name`, `description`, and `category`. Append a new `App` object to the list. Finally, print the list.

```python
# 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)
```

▼ **Explaining the Output**

The output from the above `print` statement is a list of elements that look something like this:

```
<app.App object at 0x7f035a7494a8>
```

This is how Python represents an object. It is an `App` object, which comes from the `app` module. The long string of numbers and letters is the location in memory where the object is stored (your memory locations will be different). If you see a list of these, then your code is working properly.

## Interacting with the Objects

Now that there is a list of objects, we can manipulate each object by iterating through the list. We no longer need the print statement in our program. Replace it with a for loop. On each iteration, call the `display` method.

```python
# 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()
```

challenge

## Try these variations:

- Call the `display` method on the third app.

  ▼

  **Solution**

  Normally, you would use a variable when instantiating an object. In this case, however, objects need to be referenced by the index in a list. Indexes start counting at 0, so the third element would be:

  ```
  apps[2].display()
  ```

- Call the `display` method for all objects that are have “social media” as the `category` attribute.

  ▼

  **Solution**

  Iterate over the list and use a conditional to determine if the `category` attribute is “social media”. If true, call the `display` method.

  ```
  for app in apps:
    if app.category == 'social media':
      app.display()
  ```

# Composition

## Composition

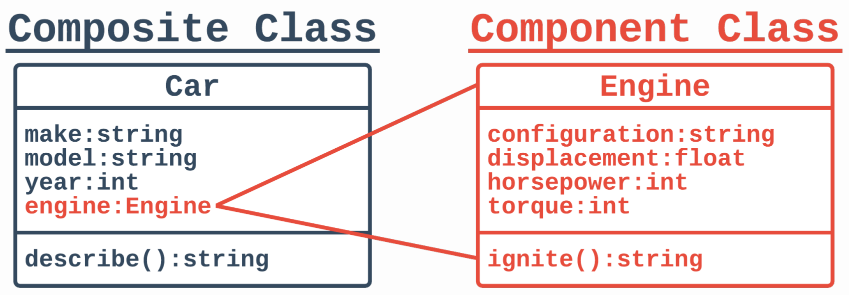
Composition is a way to make a functional whole out of smaller parts. If you were to create a `Car` class, this would start out as a simple exercise. Every car has a make, a model, and a year it was produced. Representing this data is simple: two strings and an integer.

```
class Car:
  def __init__(self, make, model, year):
    self.make = make
    self.model = model
    self.year = year

  def describe(self):
    print(f'{self.year} {self.make} {self.model}')

my_car = Car("De Tomaso", "Pantera", 1979)
my_car.describe()
```

The `Car` class, however, is missing an important component: the engine. What data type would you use to represent an engine? Creating another class is the best way to do this. Modify the `Car` class so that it has an `engine` attribute. Then create the `Engine` class with attributes for configuration (V8, V6, etc.), displacement, horsepower, and torque. Finally, add the `ignite` method to the class. Now, you have to instantiate an `Engine` object and pass it to the `Car` class.

```
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()
```

The combination of the `Car` class and the `Engine` class lead to a better representation of an actual car. This is the benefit of object composition. Because the `Engine` class is a part of the `Car` class, we can say that the `Engine` class is the component class and the `Car` class is the composite class.



.guides/img/advanced/composite_component challenge

## Try these variations:

- In the `Car` class, add the `start` method then have the last line of the script call `start` instead of `ignite`:

```
  def start(self):
    self.engine.ignite()

# Code for Engine class

my_engine = Engine("V8", 5.8, 326, 344)
my_car = Car("De Tomaso", "Pantera", 1979, my_engine)
my_car.start()
```

- In the `Engine` class, create the `info` method then call `info` instead of `describe`:

```
  def info(self):
    self.describe()

my_engine = Engine("V8", 5.8, 326, 344)
my_car = Car("De Tomaso", "Pantera", 1979, my_engine)
my_car.engine.info()
```

▼ **Why is there an error?**
Composition is a one-way street. The composite class (the `Car` class) has access to all of the attributes and methods of the component classes (the `Engine` class). However, component classes cannot access the attributes and methods of the composite class.

## Composition versus Inheritance

Assume you have the class `MyClass`. You want to use this class in your program, but it is missing some functionality. Do you use inheritance and extend the parent class, or do you use composition and create a component class? Both of these techniques can accomplish the same thing. This is a complex topic, but you can use a simple test to help you decide. Use inheritance if there is an “is a” relationship, and use composition if there is a “has a” relationship.

For example, 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.

# Representing an Object as a String

## The `__str__` Method

When you print out the instance of a user-created class, Python returns only the class name and its location in memory.

```
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

my_dog = Dog('Rocky', 'Pomeranian')
print(my_dog)
```

This is not very helpful. That is why we have seen code examples where classes have a method called `describe` or `display` that print out a better description of the object. However, the pythonic of representing an object in string form is to use the `__str__` method.

```
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)
```

The `__str__` method is used to make a human-readable string representation of a user-defined object. You may have noticed that the `print` function automatically calls the `__str__` method. While `my_dog.__str__()` is valid Python code, the `__str__` method is called with the `str()` helper function. So `my_dog.__str__()` becomes `str(my_dog)`.

## The `__repr__` Method

Python also has the `__repr__` method, which is used to create a slightly different string version of an object. The `__str__` method is for other humans, while the `__repr__` method is used to provide a very precise object definition. This definition is often used by other developers for debugging purposes. For example, the `__str__` method does not indicate the class. Most people would not need to know this, but a developer would. The `__repr__` method often is a single line of Python code that indicates the class and attributes of the object. **Note**, you need to use the `repr()` helper method with `print` otherwise you will invoke the `__str__` method.

```
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

  def __str__(self):
    return f'{self.name} is a {self.breed}'

  def __repr__(self):
    return f'Dog({self.name}, {self.breed})'

my_dog = Dog('Rocky', 'Pomeranian')
print(repr(my_dog))
```

challenge

## Try these variations:

- Erase the `__str__` method and change the print statement to `print(my_dog)`.

```python
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

  def __repr__(self):
    return f'Dog({self.name}, {self.breed})'

my_dog = Dog('Rocky', 'Pomeranian')
print(my_dog)
```

- Erase the `__repr__` method and change the print statement to `print(repr(my_dog))`.

```python
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

my_dog = Dog('Rocky', 'Pomeranian')
print(repr(my_dog))
```

▼ **Why are there no errors?**

In the first variation, the `print` method calls the **`str`** method, but the `Dog` class did not define the **`str`** method. In the second variation, the `print` method calls the **`repr`** method, but the `Dog` class did not define the **`repr`** method. How come there are no error messages? For the examples on this page you are not defining the **`str`** and **`repr`** methods; instead you are overriding them. These methods have default behavior. The **`str`** calls the **`repr`** by default, and the **`repr`** prints the class name and memory address by default.

## Printing a List of Objects

Create a list of `Dog` objects. Be sure to override the `__str__` method and then print the list.

```
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

  def __str__(self):
    return f'{self.name} is a {self.breed}'

dogs = []
dogs.append(Dog('Rocky', 'Pomeranian'))
dogs.append(Dog('Bullwinkle', 'Labrador Retriever'))
print(dogs)
```

When you have a collection of user-defined objects, the `print` function does not return output from the `__str__` method. Instead, it defaults to the `__repr__` method. Replace the `__str__` method with a `__repr__` method and print the list again.

```
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed

  def __repr__(self):
    return f'Dog({self.name}, {self.breed})'

dogs = []
dogs.append(Dog('Rocky', 'Pomeranian'))
dogs.append(Dog('Bullwinkle', 'Labrador Retriever'))
print(dogs)
```