Classes in Python

By Nicola Romanò - last updated on 25 July 2024

This document will provide an introduction to the use of classes in Python.

Introduction

Sometimes we find ourselves in a situation where Python's basic data types (e.g. int, float, bool) are not enough, and we need to create our own data types.

Classes are a data structure used to represent some entity. They are central to a programming paradigm called **object oriented programming** (OOP), which is used in many programming languages.

Why using classes?

Let's make a practical example, to understand why we need classes, first of all.

You want to create a software to manage books in a book shop. You would need to store information such as the book's title, author, price, etc.
Using Python's basic data types you could do it like this:

```
titles = ['The Lord of the Rings', 'Moby Dick', 'The Great Gatsby']
authors = ['J. R. R. Tolkien', 'Herman Melville', 'F. Scott Fitzgerald']
prices = [19.99, 8.99, 11.99]
```

This is going to be very messy very soon, and it is not practical if we want to add new characteristics to our books, such as the number of pages.

There are slightly better ways to do this, such as using dictionaries, but wouldn't it be better to just have a *Book* variable type? That is exactly what classes are for!

We can define a new **class** called `Book`, with all of the necessary information to describe a book. A good metaphore for a class is a *blueprint* that tells Python how to structure this new data type.

Once we define a class, we can create multiple instances of it called **objects**; an object can contain variables, called **attributes**, and functions, called **methods**.

Creating your first class

So, enough talk... let's create our `Book` class!

This class will have four attributes (variables): `title`, `author`, `year`, `price`.

We can create a class in Python by using the `class` keyword.

```
class Book:
    """
    A class representing a book.
    """

    def __init__(self, title, author, year, price):
        """
        Initialize the attributes of the class.
        Params:
            title (str): The title of the book.
            author (str): The author of the book.
            year (int): The year the book was published.
            price (float): The price of the book, in £.
        """

        self.title = title
        self.author = author
        self.year = year
        self.price = price
```

You can see that we have created a function called `__init__`. This is a special function called a **constructor** and it is used to initialize the attributes of the class. When we create an instance of the class, the constructor will be called automatically.

Note how we have used the `self` keyword. This is a special keyword that is used to access the attributes of the class. We will see more about this in the next section.

Note how I used docstrings to describe the class. While our `Book` class is very straightforward, it is good practice to get into the habit of always documenting your classes and functions.

Let's now create two instances of the class.

```
book1 = Book("Moby Dick", "Herman Melville", "1851", 20.50)
book2 = Book("The Great Gatsby", "F. Scott Fitzgerald", "1925", 15.0)
```

When we call `Book` we are automatically calling the constructor, passing it the various parameters for that specific book.

Note that the constructor takes a `self` parameter, however we have not used it in the code. This is because Python invisibly passes the `self` parameter to the constructor, so you don't have to worry about it!
We can call the contrsuctor multiple times, and each time it will create a new **independent** instance of the class (a Book object).

Accessing attributes

Accessing attributes is done by using the `.` operator.

```
print(book1.title)
print(book2.author)
```

Although we can directly modify attributes, this is not always a good idea.

For example, this works:

```
book1.price = 10.0
print(book1.price)
```

But nothing prevents us from doing:

```
book1.price = -15.30
```

which obviously will not make sense! It is better practice to write specific methods (the functions in a class) to set and retrieve attributes.

Methods

Let's create a method to set the book's price.

We will add the following to our class definition:

```
def set_price(self, price):
    """
    Set the price of the book.
    Params:
        price (float): The price of the book, in £.
                    Must be positive.
    """

    if price > 0:
        self.price = price
    else:
        print("The price must be positive!")
```

Note how we need to pass `self` as the first parameter to the method, although we never explicitely pass it.

Let's create another method to print information about the book. We modify our class definition as follows:

```
def print_info(self):
    """
```

```
        Prints information about the book.
        """
        # The .2f format specifier formats a float to 2 decimal places.
        print(f"{self.title} by {self.author}, {self.year}. Price £{self.price:.2f}")
```

We can now do:

```
book1.print_info()
book2.print_info()
```

Exercise 1

1. You want to keep track of the stock of books in the library. Add a method to the class called `add_stock` that takes a number as an argument and adds it to the `stock` attribute.
2. Add a method called `remove_stock` that takes a number as an argument and removes it from the `stock` attribute. This should check that the number to be removed is not greater than the current stock, or print an error message otherwise.
3. Finally, add a method called `print_stock` that prints the current stock.

Congratulations! You have written your first class!

The Book class can now be used as a new type, even inside another class!

For example, we can create a new class called `Bookshop` that will contain a list of `Book` objects.

```
class Bookshop:
    """
    A class representing a bookshop.
    """

    def __init__(self, name):
        """
        Initialize the attributes of the class.
        Params:
            name (str): The name of the bookshop.
        """

        self.name = name
        self.books = []

    def add_book(self, book):
        """
        Add a book to the bookshop.
```

```
        Params:
            book (Book): The book to add.
        """

        self.books.append(book)
```

We can then add books to the bookshop:

```
bookshop = Bookshop("Books for everyone")
bookshop.add_book(Book("Moby Dick", "Herman Melville", "1851", 20.50))
bookshop.add_book(Book("The Great Gatsby", "F. Scott Fitzgerald", "1925", 15.0))
bookshop.add_book(Book("The Hobbit", "J. R. R. Tolkien", "1937", 10.0))

for book in bookshop.books:
    book.print_info()
```

In reality you would read the names etc from a database or a file, but that is beyond the scope of this tutorial.

Inheritance

Inheritance is a way to create new more *specialised* classes from existing classes.

Let's say you have a class `Person` that has the attributes `name` and `age`.
You can imagine that for some people it might make sense to have a `company` and a `salary` attributes; for others it might instead have more sense to have a `school` attribute, if they are students.

So, certain attributes (and this applies to methods as well) can be shared by all people, but others can be unique to each "subtype" of person.

You can create a class `Student` that inherits from `Person` and has the additional attribute `school`, representing the school the student attends. You can also create a class `Worker` that inherits from `Person` and has the additional attribute `company`, representing the company the worker works for.

This is quite a hefty chunk of code, so take your time to understand it.

```
class Person:
    """
    A class representing a person.
    """

    def __init__(self, name, age):
        """
        Initialize the attributes of the class.
        Params:
            name (str): The name of the person.
```

```
            age (int): The age of the person.
        """

        if age <= 0:
            print("The age must be positive!")
        else:
            self.name = name
            self.age = age

    def print_info(self):
        """
        Prints information about the person.
        """
        print(f"{self.name} is {self.age} years old.")

class Student(Person):
    """
    A class representing a student.
    Inherits from Person.
    """

    def __init__(self, name, age, school):
        """
        Initialize the attributes of the class.
        Params:
            name (str): The name of the student.
            age (int): The age of the student.
            school (str): The school the student attends.
        """

        # Pass name and age to the Person class
        super().__init__(name, age)
        self.school = school

    def print_info(self):
        """
        Prints information about the student.
        """

        # Call the Person class print_info method
        super().print_info()
        print(f"{self.name} attends {self.school}.")


class Worker(Person):
    """
    A class representing a worker.
    Inherits from Person.
    """

    def __init__(self, name, age, company):
        """
```

```
        Initialize the attributes of the class.
        Params:
            name (str): The name of the worker.
            age (int): The age of the worker.
            company (str): The company the worker works for.
        """

        # Pass name and age to the Person class
        super().__init__(name, age)
        self.company = company

    def print_info(self):
        """
        Prints information about the worker.
        """

        # Call the Person class print_info method
        super().print_info()
        print(f"{self.name} works for {self.company}.")
```

Finally let's try it and see if this works!

```
person1 = Person("John", 30)
person1.print_info()

student1 = Student("Weiwei", 20, "MIT")
student1.print_info()

worker1 = Worker("Yuki", 40, "Olympus")
worker1.print_info()
```

You can see that Python automatically chooses which `print_info()` method to call based on the type of object.

Exercise 2

1. What do you think will happen if you created a class inheriting from `Person` that did not have its own `print_info()` method?
2. Try to implement that!

Now you should know the basics of how to use classes in Python! Using classes can feel a little bit odd at the beginning, but once you get used to it, you'll find it really useful!