Print to PDF

Week 10: Classes 2

Contents

Important

First, `download_week10.py` and run the script in `VSCode`. You may need to confirm the download in your browser.

The script should end with `Successfully completed!`. If you encounter an error, follow the instructions in the error message. If needed, you can proceed with the exercises and seek assistance from a TA later.

This week, we delve into more advanced aspects of classes. We will explore three key topics to help us better understand classes:

- **Inheritance**: Python supports a concept called class inheritance. It allows us to extend existing classes by adding new methods and attributes without starting from scratch. We will work with various examples to create a new class that inherits from an existing one.
- **Operator Overloading**: To truly embrace Object-Oriented Programming, we must discuss operator overloading. This means defining how operators (`+`, `-`, `>`, etc.) should behave for instances of your classes. In other words, you can make your objects support standard operators, and we'll explore this concept further.
- **Everything in Python is from a class**: In Python, even basic data types like strings and integers are classes. That is, any string is an instance of the `str` class. This is how strings can have methods like `.lower()`, and `.join()`, because they are class methods. We will dive deeper into understanding and manipulating classes and their methods.

By now, you should be familiar with the basic concepts of classes and how to create them.

Exercise 10: Inheritance and classes

Often, we need to create a class similar to an existing one but with some differences. Now, you could rewrite the entire class, but that takes a lot of time and effort, and it's not very efficient. Instead, you can efficiently build upon an existing class by using inheritance. Again, most examples provided here are simple enough that inheritance seems like overkill, but it is a very useful tool when working with more complex classes.

Let's look at an example of this:

```
>>> class Pet:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def get_description(self):
...         return f"{self.name} is {self.age} years old"
...
```

Now, we would like to create a `Dog` class, which works in almost the same way as `Pet`, but also stores which tricks the dog is able to perform. We *could* do this by creating an entirely new class:

Skip to main content

```
...         self.name = name
...         self.age = age
...         self.tricks = tricks
...     def get_description(self):
...         return f"{self.name} is {self.age} years old"
...
>>> my_dog = Dog("Fido", 3, ["sit", "stay", "fetch"])
>>> my_dog.get_description()
'Fido is 3 years old'
```

Instead of this, we can inherit from the `Pet` class. This is done by specifying which class we want to inherit from when defining `class Dog(Pet)` class. Here is a full example:

```
>>> class Dog(Pet):
...     def __init__(self, name, age, tricks):
...         super().__init__(name, age) # Call the Pet.__init__() method.
...         self.tricks = tricks
...
>>> my_dog = Dog("Fido", 3, ["sit", "stay", "fetch"])
>>> my_dog.get_description()
'Fido is 3 years old'
```

Note

Here, the `__init__` function uses `super()` to call the `__init__` function of the parent class. Thus, we don't have to reimplement the entire `__init__` function in the subclass.

The built-in function `super()` tells Python to access the parent class (the class the current class inherits from). It is a way to access and use the parent class's functionality from the new class.

We can even use `super()` to extend `get_description` to include the tricks the dog knows, without having to rewrite the entire method.

```
>>> class Dog(Pet):
...     def __init__(self, name, age, tricks):
...         super().__init__(name, age) # Call the Pet.__init__ method.
...         self.tricks = tricks
...     def get_description(self):
...         return super().get_description() + ", and knows these tricks: " + ", ".join(self.tricks)
...
>>> my_dog = Dog("Fido", 3, ["sit", "stay", "fetch"])
>>> print(my_dog)
<__console__.Dog object at 0x7f2dc2267be0>
>>> my_dog.get_description()
'Fido is 3 years old, and knows these tricks: sit, stay, fetch'
```

Note that we can overwrite the existing method while still using the `super()` keyword to access the overwritten method in the parent class.

Finally, we can implement **operator overloading**. This means defining how operators (`+`, `-`, `>`, etc.) should behave for instances of our classes. In other words, you can make your objects support standard operators, which you will get to do in the exercise. For now, let's look at what happens if we rename our `get_description` method to `__str__`:

```
>>> class Dog(Pet):
...     def __init__(self, name, age, tricks):
...         super().__init__(name, age) # Call the Pet.__init__ method.
...         self.tricks = tricks
...     def __str__(self):
...         return super().get_description() + ", and knows these tricks: " + ", ".join(self.tricks)
...
>>> my_dog = Dog("Fido", 3, ["sit", "stay", "fetch"])
>>> print(my_dog)
Fido is 3 years old, and knows these tricks: sit, stay, fetch
```

As we can see, when we now try to `print` `my_dog`, it actually calls the `__str__` method of the class and prints the string returned by that method. This is a very useful feature to have a user-friendly way of printing the contents of your custom classes.

Skip to main content

Exercise 10.1: Inheritance - ExtendedString.to_snake_case()

Exercise 10.1: snake case

Note

snake_
where
unders
case. T
progra
functio

You will now expand on an already existing class. Recall that all built-in data types in Python are actually classes, including `str`. For this exercise, you will be asked to expand on the `str` class, by creating the class `ExtendedString`, which adds more functionality by including an additional method compared to the `str` class. You should implement the method `to_snake_case()`, which takes in a `str` and replaces all space characters (' ') with the underscore character (`_`), while also turning all upper-case letters to lower-case. Therefore, the algorithm should return the `snake_case` version of the string as illustrated below in an example:

Tip

Tip

To illustrate how the function can be used, have a look at the following example:

```
>>> text = ExtendedString("ThiS is a NoN snaKE teXT.")
>>> text.to_snake_case()
'this_is_a_non_snake_text.'
>>> text.split()
['ThiS', 'is', 'a', 'NoN', 'snaKE', 'teXT.']
```

Note that since `ExtendedString` inherits from `str`, you can use all the methods of the `str` class on an instance of `ExtendedString`, as we showed in the above code by using `.split()` on the `text` variable.

To be able to test your method, add it to the class in the file `cp/ex10/extended_string.py`.

class **cp.ex10.extended_string.ExtendedString**

Your expanded version of the python str class.

to_snake_case()

Return the 'snake_case' version of a string with lower-case characters and underscores instead of spaces.

Return type:
`str`

Returns:
The snake_case version of the input text.

Exercise 10.2: Inheritance - ExtendedString.word_count()

Exercise 10.2: word_count

We will now add another costum method to the `str` class. The goal of this exercise is to expand your implemented version of the `ExtendedString` class, by adding the `word_count()` method. This method is supposed to take a text formatted in a string and return the number of words within the text.

Tip

Implemented, you should be able to use the method like this:

```
>>> y = ExtendedString("How are you?")
>>> y.word_count()
3
>>> x = ExtendedString("This sentence contains five words.")
>>> x.word_count()
5
```

Skip to main content

class cp.ex10.extended_string.**ExtendedString**

Your expanded version of the python str class.

word_count()

Count the number of words contained in a string.

Return type:
`int`

Returns:
The number of words within the given text.

Exercise 10.3: Inheritance - Student

Exercise 10.3: Student

For this exercise, we will expand the `Person` class that was defined in the previous week. Recall that the class looks like this:

```
>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...     def get_full_name(self):
...         full_name = self.first_name + ' ' + self.last_name
...         return full_name
...
```

You should now create a `Student` class that inherits from the `Person` class. The new feature of the `Student` class is, that it comes with an additional attribute, which is the degree the student is studying `BSc` (bachelors) or `MSc` (masters).

Furthermore, the `Student` class should have an additional method `remaining_ECTS`. This method takes as input how many ECTS the student has completed so far and returns how many more ECTS the student has to complete before graduation. For a `BSc` the total number of ECTS required is 180, while for a `MSc` it is 120.

Tip

You should be able to use the new class as follows:

```
>>> the_student = Student('Lars', 'Larsen', 'BSc')
>>> the_student.remaining_ECTS(60) # 60 ECTS completed, 120 remaining
120
```

To be able to test your class, add it to the class in the file `cp/ex10/student.py`.

class cp.ex10.student.**Student**(*first_name*, *last_name*, *degree*)

An extended version of the Person class which also includes the degree.

__init__(*first_name*, *last_name*, *degree*)

Initialize the Student instance.

Parameters:
- **first_name** (`str`) – The student's first name.
- **last_name** (`str`) – The student's last name.
- **degree** (`str`) – The degree of the student ('BSc' or 'MSc').

remaining_ECTS(*ECTS_to_date*)

Calculate the remaining number of ECTS points before graduation.

Parameters:

Skip to main content

Return type:

`int`

Returns:

The number of ECTS points remaining before graduation.

Exercise 10.4: Operator Overloading - Vector Addition

Exercise 10.4: Operator Overloading

Now, let us dive into the topic of operator overloading. In Python, operator overloading is a concept that allows you to modify the behavior of standard operators for instances of classes. This flexibility enables you to write code specifically designed to meet the requirements of your problem.

Note

To change the way basic math operators work in Python, you should define methods with specific names that Python recognizes. Here's the naming convention for overriding these operators:

- For the addition operator `+` the method should be named `__add__`.
- For the subtraction operator `-` the method should be called `__sub__`.
- For the multiplication operator `*` the method should be called `__mul__`.

Keep these naming conventions in mind when implementing the 3 following exercises.

If you encounter any difficulties while implementing this, you can have a look at the `Time` class implemented section 17.7 in [Dow16].

To better understand this concept, let us revisit the `Vector` class we worked with last week.

```
>>> class Vector:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
```

The goal of this exercise is to be able to add two vectors elementwise using the `+` operator. The addition of two vectors is defined as: $\vec{a} + \vec{b} = (a_x + b_x, a_y + b_y)$.

When implemented, your function should work like this:

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(3, 4)
>>> v3 = v1 + v2
>>> (v3.x, v3.y)
(4, 6)
```

To be able to test your class, add it to the class in the file `cp/ex10/vector.py`.

class cp.ex10.vector.**Vector**(*x*, *y*)

A class that represents a Vector, defined by the endpoint (x, y).

__add__(*other*)

Define addition for two Vector-objects.

Parameters:

other – The second vector of the addition operation

Returns:

The vector sum

Skip to main content

Exercise 10.5: Operator Overloading - Vector Subtraction

Exercise 10.5: Operator Overloading

Now you should implement subtraction of two vectors using the `-` operator, which is defined as: $\vec{a} - \vec{b} = (a_x - b_x, a_y - b_y)$.

When implemented, your function should work like this:

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(3, 4)
>>> v3 = v1 - v2
>>> (v3.x, v3.y)
(-2, -2)
```

To be able to test your method, add it to the class in the file `cp/ex10/vector.py`.

class cp.ex10.vector.**Vector**(*x*, *y*)

A class that represents a Vector, defined by the endpoint (x, y).

__sub__(*other*)

Define subtraction for two Vector-objects.

Parameters:

other – The second vector for the subtraction operation

Returns:

The vector subtraction

Exercise 10.6: Operator Overloading - Dot Product

Exercise 10.6: Operator Overloading

And finally add the multiplication (dot product) of two vectors. The dot product is defined as: $\vec{a} \cdot \vec{b} = a_x * b_x + a_y * b_y$

When implemented your function should work like this:

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(3, 4)
>>> v1 * v2
11
```

To be able to test your method, add it to the class in the file `cp/ex10/vector.py`.

class cp.ex10.vector.**Vector**(*x*, *y*)

A class that represents a Vector, defined by the endpoint (x, y).

__mul__(*other*)

Define inner product for two Vector-objects.

Parameters:

other – The second vector for the dot product

Returns:

Dot product result

Exercise 10.7: Shopping Cart - Item

The last two exercises for this week will focus on a project where we'll be developing a virtual shopping cart for an online store. To achieve this, start with the following:

Note

The fo
Projec
link cc
includ

Skip to main content

- `name`, indicating the name of the object.
- `quantity`, indicating how many of those items one has.
- `price`, indicating the price (per item) for the specific object.

Additionally, the `Item` class should implement the methods `__lt__`, `__gt__`, and `__eq__`, which correspond to the operators `<`, `>`, and `==` respectively. The methods should take another item as input and compare the two items based on their total cost (`quantity * price`). The methods should return a boolean value indicating the result of the comparison.

The following example illustrates the expected `Item` functionality.

```
>>> item1 = Item("Laptop", 1, 800)
>>> item2 = Item("Phone", 1, 400)
>>> item3 = Item("Tablet", 2, 400)
>>> item1.name
'Laptop'
>>> item1.quantity
1
>>> item1.price
800
>>> item1 < item2
False
>>> item2 > item3
False
>>> item1 == item3
True
```

Add your class to the file `cp/ex10/shopping_cart.py` to get it graded for the project.

class cp.ex10.shopping_cart.**Item**(*name*, *quantity*, *price*)

A class to represent an inventory item.

__init__(*name*, *quantity*, *price*)

Initialize a new Item object.

Parameters:

- **name** (`str`) – The name of the item.
- **quantity** (`int`) – The quantity of the item.
- **price** (`float`) – The price of the item.

__lt__(*other*)

Define the `<` operator on the Item class.

Parameters:

other – The second item that we are comparing to.

Returns:

a boolean variable indicating if the "less than" condition is satisfied.

__gt__(*other*)

Define the `>` operator on the Item class.

Parameters:

other – The second item that we are comparing to.

Returns:

a boolean variable indicating if the "greater than" condition is satisfied.

__eq__(*other*)

Define the `==` operator on the Item class.

Parameters:

other – The second item that we are comparing to.

Returns:

a boolean variable indicating if the "equals" condition is satisfied.

Note

Again
Pytho
with s
recogr
conve
operat

- F
n
- F
t
- F
c

Skip to main content

Exercise 10.8: Shopping Cart Inventory

Build a custom `Inventory` class responsible for storing and managing different items that belong in the shopping cart. You should inintialize the class with an empty list of items. This class should include the following methods:

- `add_item`: Adds an item to the cart.
- `calculate_total_value`: Calculate the total value of the items in the cart.
- Operator overloading for comparing the total price of purchased items using the "less than", "greater than" and "equals" operators, just like in the previous exercise but for all items in the Inventory.

The following example illustrates the expected `Item` and `Inventory` functionality.

```
>>> inventory1 = Inventory()
>>> item1 = Item("Laptop", 10, 800)
>>> item2 = Item("Phone", 20, 400)
>>> item3 = Item("Tablet", 15, 300)
>>> inventory1.add_item(item1)
>>> inventory1.add_item(item2)
>>> inventory1.add_item(item3)
>>>
>>> inventory2 = Inventory()
>>> item4 = Item("Laptop", 2, 850)
>>> item5 = Item("Phone", 5, 420)
>>> inventory2.add_item(item4)
>>> inventory2.add_item(item5)
>>> inventory1.calculate_total_value()
20500
>>> inventory2.calculate_total_value()
3800
>>> inventory1 > inventory2
True
>>> inventory1 < inventory2
False
>>> inventory1 == inventory2
False
```

Add your class to the file `cp/ex10/shopping_cart.py` to get it graded for the project.

class **cp.ex10.shopping_cart.Inventory**

A class to represent an inventory of items.

__init__()

Initialize a new Inventory object with an empty list of items.

add_item(*item*)

Add an item to the inventory.

Parameters:

item (`Item`) – The Item object to be added.

calculate_total_value()

Calculate the total value of the inventory.

Return type:

`float`

Returns:

The total value of all items in the inventory.

__lt__(*other*)

Compare two inventories based on their total values.

Parameters:

other – The other Inventory object to compare with.

Return type:

`bool`

Returns:

Skip to main content

`__gt__(other)`

Compare two inventories based on their total values.

Parameters:

other – The other Inventory object to compare with.

Return type:

`bool`

Returns:

True if the current inventory's total value is greater than the other's; otherwise, False.

`__eq__(other)`

Compare two inventories based on their total values.

Parameters:

other – The other Inventory object to compare with.

Return type:

`bool`

Returns:

True if the current inventory's total value is equal to the other's; otherwise, False.