Introduction to Python Day 7

Verjinia Metodieva and Daniel Parthier

2025-04-01

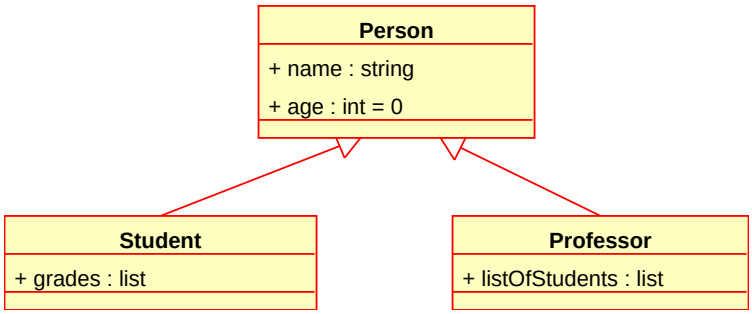
Classes

- Classes are objects
- Can combine data and functions
- Can create complex data structures
- Helps you to organise your code and data

Where are they useful?

- You want to combine different structures
- You need to add information to your data
- You want to use/reuse specific functions with your data

Class structure and relationship



General structure

```
class ClassName:
    def __init__(self, parameters):
        # Constructor
        self.attribute = parameters

    def method(self, parameters):
        # Method
        return self.attribute + parameters
```

- `ClassName` is the name of the class
- `__init__` is the constructor
- `self` is the instance of the class
- `parameters` are the function inputs
- `self.attribute` is the attribute of the class
- `method` is a method of the class
  - methods are functions that belong to the class
- `return` is the output of the method

The concept of `self`

- `self` is a reference to the current instance of the class
- Accessing `self` with dot notation allows you access/modification of the class
  - `self.attribute`
- A method of a class might require to access the class itself
  - `self` is passed as first argument to the method
  - `self` could be also called `this`, `cls`, or any other name
  - It is convention to use `self` in Python

What could a class look like?

```
class Experiment:
    def __init__(self, name, date):
        self.name = name
        self.date = date
        self.data = []

    def add_data(self, data):
```

```
        self.data.append(data)

    def get_data(self):
        return self.data
```

How to use them?

```
test_experiment = Experiment("LTP", "2025-04-15")
type(test_experiment)
test_experiment.name
test_experiment.date
test_experiment.add_data([1.41, 1.38, 1.39])
test_experiment.get_data()
```

```
__main__.Experiment
```

```
'LTP'
```

```
'2025-04-15'
```

```
[[1.41, 1.38, 1.39]]
```

- Similar to what we saw before:
  - `test_experiment` is an instance of the class `Experiment`
  - `name` and `date` are attributes of the class (dot notation)
  - `add_data` and `get_data` are methods of the class (dot notation)

How to use them?

- We can now write functions which use the class as input

```
def average_LTP(experiment: Experiment) -> float:
    """
    Calculate the average of the data in an Experiment object.
    Parameters:
    experiment (Experiment): The Experiment object containing data.
    Returns:
    float: The average of the data.
    """
    import numpy as np
```

```
    if not isinstance(experiment, Experiment):
        raise TypeError("Input must be an Experiment object.")

    return np.mean(experiment.data)
```

- Function requires the `Experiment` class as input
- Checks if the input is of the correct type
- Calculates the average of the data in the `Experiment` object

Notice we can use type hints to specify the input and output type of the function. In our case the input is an `Experiment` object and the output is a float. This is not mandatory, but it helps to understand the function better. You can use the import statement to import the numpy library inside the function.

What is an advantage of using a class?

- We could specify sensible functions to use with the object
- An external function could require to use multiple inputs
  - A class can combine all the inputs into one object
- Well defined classes can be reused for different functions
- A lot of information can be stored in one object
- Classes can be dynamic

Example for an event class

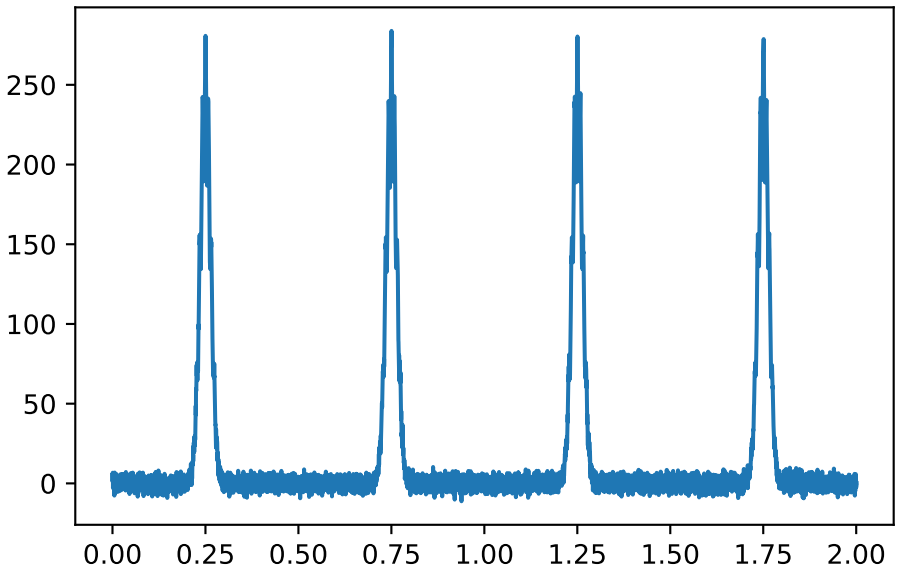
```
time = np.linspace(start=0, stop=0.5, num=2000)
mu, sigma = 0.25, 0.01
sinewave = np.sin(time * 250 * np.pi)
gaussian = (1 / (np.sqrt(2 * np.pi * np.square(sigma))) *
            np.exp(-(np.square(time - mu) /np.square(2 * sigma))))
ripple = gaussian * sinewave
```

- Recall from previous lessons

Example for an event class

```
np.random.seed(0)
trace = ripple + gaussian*6
# repeat the trace 4 times
```

```
# add noise
trace = np.tile(trace, 4)
trace += np.random.normal(0, 3, size=trace.shape)
# add time to the trace
time = np.arange(start=0, step=time[1], stop=len(trace)*time[1])
plt.plot(time, trace)
plt.show()
```



- We can use the `trace` and `time` to create an event class

In class coding

- Create an `Event` class

Now let's make our own

- Make a class!