class

> Classes provide a means of bundling data and functionality together.
> Creating a new class creates a new type of object, allowing new instances of that type to be made.

A class can have variables attached to it, it can also have functions attached to it. If you crate a new instance of that class, it will also have variables and functions attached to it.
Classes are behind the concept of **Object Oriented Programming**: an instance of a class is also called an `object`.

You actually already used classes: `integer`, `string` or `list` are actually `python` classes.
All `lists` are instances of the `list` class contained within `python`.

Examples help.

class

Let's build our first class. Building classes without any thinking behind it is not useful. In this case, we are going to create a class that represents cats.
Let's define our class:

```
class Cat:
        pass
```

That's it ! This is the simplest class you can make ! It is useless, but it shows how you can initialize a class. Notice that everything within the class must be indented.

Later on during the code, or even from another file, you can assign a new `cat` object to the `floki` variable doing:

```
floki = Cat()
```

class

Ok, now let's improve our class a bit. To initialize a class (let's be honest, our previous class was really useles), you always need a `__init__` function, like this:

```python
class Cat:

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

An `__init__` method always takes `self` as an input, `self` refers to the class itself. Any function which is part of the class (meaning that it will be able to access variables and functions from the class) must take `self` as an input.
Here, the class is initialized by providing a name and age for our cat. To initialize this class, you could use:

```python
floki = Cat("floki", 3)
```

class

```
floki = Cat("floki", 3)
```

If you then try:

```
>>> floki.name
'floki'
>>> floki.age
3
```

Notice how the `Cat` object is carrying all its variables with it, and these can be called any time. We could also have a second cat object:

```
louise = Cat("louise", 2)
```

We could then retrieve information about louise using: `louise.name` and `louise.age`
We can actually retrieve these variables because they were defined using `self.name = name`. If a variable is defined within the class without a `self` it will not be accessible from outside the class.

class

Alright, our classes can now have variables that we can access from outside the class.
Let's try to add a class function:

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def hungry(self):
        print("Miaou") #these cats are moroccan they dont say meow
```

Notice how our function takes a `self` as an input even if there is no input. Since it is a class function it needs to input the `object` itself.
Now if we initialize the class again, we should be able to trigger this class function:

```
>>> floki = Cat("floki", 3)
>>> floki.hungry
'Miaou'
```

class

A function can also be triggered when the class initializes:

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.hungry()

    def hungry(self):
        print("Miaou") #these cats are moroccan they dont say meow
```

Notice how we are calling the function with a self before it since it is a function within the class. This time, if you create a new instance of a cat object:

```
>>> floki = Cat("floki", 3)
'Miaou'
```

The `hungry()` function is triggered during its initialization.

class

Don't forget, if you define a variable without a `self` before it, it will only be accessible within the class

```
class Cat:
    def __init__(self, name, age):
        name = name
        age = age
```

For example, this time, you will not be able to print name and age after initializing your class.

import

A few additions about import. We have explored in functions how imports can be used to import functions or classes from other files.

Let's say we have a folder with two files:

```
ilyass@tx1:~/ex_imports$ ls
myfunctions.py  myscript.py
```

The file `myfunctions.py` contains a function called `hellofunc()` that prints `"Hello"` when called:

```
def hellofunc():
        print("hello")
```

The file `myscript.py` contains an import and then a call to the `hellofunc()` function:

```
from myfunctions import hellofunc

hellofunc()
```

import

Since both files are in the same folder, the `from myfunctions import hellofunc` will try to import a python package installed on your system which is called `myfunctions` and that contains a `hellofunc` function, but since there is no such package it will find the file in the same folder called `myfunctions.py` and it will import `hellofunc`.

If you execute the `myscript` it will thus successfully print `'hello'`:

```
ilyass@tx1:~/ex_imports$ python myscript.py
hello
```

It is a convenient way to store some functions you might need in several projects.
You can import functions and classes from other files using this method.
It is also possible to import functions and classes stored in folders within your project folder, it requires respecting a certain folder and file structure. We will not be exploring these structures in this intro class.