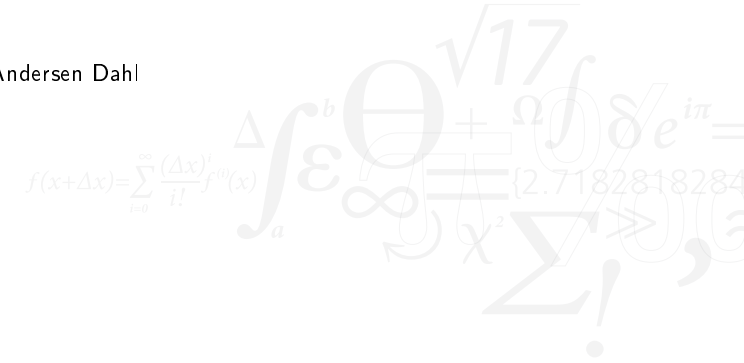
Lecture 9: Classes, part 1

Morten Rieger Hannemose, Vedrana Andersen Dahl

Fall 2023



Today's lecture

1. An introduction to OOP (15 min)
2. A coding introduction to OOP (45 min)

OOP: object-oriented programming

- OOP is a way of structuring programs where properties and behavior are bundled into classes consisting of individual objects.
- The true value of OOP is visible in bigger projects

Recall things we've seen

```
f = open('my_file.txt', 'w')  # file object
f.write('This is a new file\n')  # file method
f.close()
```

```
my_string = 'Hello world!'  # string object
shout = my_string.upper()  # string method
```

```
my_list = ['C', 'D', 'A']  # list object
my_list.append('B')  # list method
my_list.sort()  # list method
```

```
my_dict = {'a': 'apple', b: 'banana'}  #
    dictionary object
fruit = my_dict.values()  # dictionary method
```

```
def my_function(name):
    print(f'Hello world! says {name}')
something = my_function  # function object
```

All entities in Python are objects

- More obvious: Lists, strings, dictionaries, file objects.
- Less obvious: Integers, floats, functions.
- Under the hood: Everything!

An object is the collection of data and methods that operate on those data.

Why teach OOP?

- Sometimes, you don't need to think too much about it.
- Sometimes it is important to know that you work with objects. (Examples from your possible future: pandas DataFrame, NumPy ND-array, PyTorch tensor class or nn module.)
- Sometimes, it may be useful to define your own classes.

OOP in *Think Python*

Chapter 15 You can define classes, your own data types, to create objects that represent some entity (a point, a patient, a customer, a train, an image). Such objects are mutable.

Chapter 16 You can write functions that take user-defined objects as parameters, modify objects, or return them as results.

Chapter 17 You can write methods, blocks of code similar to functions, but associated with a particular class. A special method is used when initializing (creating) objects.
You can define operators for your data types, and functions that handle different data types (polymorphism).

Chapter 18 A new class can be a modified version of an existing class (inheritance).

Week 9 covers Chapters 15 and 16, but also methods and a special method used for initialization.

A note on terminology

OOP is used in many programming languages, and terminology may vary slightly.

Type and class is the same

- In Python, a type and a class is the same. (It used to be that types are built-in, and classes user-made.)

Instance of a class

- Class is a template, prototype, blueprint, mold . . . for defining instances.
- Instance is a concrete object of a certain class.

Objects

- Term *object* and *instance* are sometimes used interchangeably. (Instances are objects and every object is an instance of some class.)
- In Python, everything is an object. (A class itself is also an object, a *class object*, of type class.)
- If it helps, you can think of the word *object* as a *something*.

In conclusion: I'll try saying class and instance to be precise.

Code shown live during lecture

I want to somehow represent the time of day, consisting of hours and minutes.

Representing time, options so far

```
# representing time using two variables
hours = 13
minutes = 8

print(f'{hours:02}:{minutes:02}')

#representing time using a dictionary
my_time = {'hours': 13, 'minutes': 28}
print(f"{my_time['hours']:02}:{my_time['minutes
    ']:02}")
print(my_time)
```

To represent time I can use two integer variables, one for hours, one for minutes. Or, I can use a built-in type, for example dictionary as shown here.

First example of a class

```
class MyTime:
    pass  # this is a placeholder for some code

my_time = MyTime()  # an instance of the class
my_time.hours = 13
my_time.minutes = 37

print(my_time.hours)
print(my_time.minutes)

other_time = MyTime()  # another instance
other_time.hours = 17
other_time.minutes = 00

print(other_time.hours)
print(other_time.minutes)
```

`MyTime` is a class, and I create two instances (objects) of this class: `my_time` and `other_time`.

We would normally assign attributes (hours and minutes) in the initialization method, but in this first example, we do it differently as we don't yet know how `self` works.

Code shown live during lecture

Objects are mutable

```
class MyTime:
    pass

my_time = MyTime()
my_time.hours = 13
my_time.minutes = 37

other_time = my_time
other_time.hours = 10

print(my_time.hours)
print(my_time.minutes)
print(other_time.hours)
print(other_time.minutes)
```

Changing `other_time` also affects `my_time` as they point to the same object.

Functions may take objects

```
class MyTime:
    pass

my_time = MyTime()
my_time.hours = 13
my_time.minutes = 37

def print_time(time):
    print(f'{time.hours:02}:{time.minutes:02}')

print_time(my_time)
```

I can write functions that take user-defined objects as arguments. Such functions may leave the objects unchanged (pure functions) or modify objects (modifiers). Here, `print_time` is a pure function. An example of a modifier would be a function incrementing the hours of the time object it received.
I can also write functions that return user-defined objects. An example would be a function that asks a user to input hours and minutes as integers, and then creates a `MyTime` instance with those values as attributes.

Code shown live during lecture

First example of a method

```
class MyTime:

    def print_time(self):
        print(f'{self.hours:02}:{self.minutes:02}')

my_time = MyTime()
my_time.hours = 13
my_time.minutes = 37

my_time.print_time()
```

Now, `print_time` is moved inside the class body. This makes it a method of the `MyTime` class.

The first argument of the method, usually called `self`, is always the instance of the class.

The argument `self` is the object the method works on. The method is called using a dot notation: object-dot-method.

`__init__` method

```
class MyTime:

    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    def print_time(self):
        print(f'{self.hours:02}:{self.minutes:02}')

my_time = MyTime(23, 4)
my_time.print_time()
```

`__init__` is a special method that gets called when an instance of the class is created.

Usually, all instance attributes get assigned in this method.

This is the first example showing the usual way of defining a class in Python: it would start with the `__init__` method.

Code shown live during lecture

Full example

```
1  class MyTime:
2
3      def __init__(self, hours, minutes):
4          self.hours = hours
5          self.minutes = minutes
6
7      def print_time(self):
8          print(f'{self.hours:02}:{self.minutes:02}')
9
10     def increment_hours(self):
11         self.hours += 1
12         if self.hours == 24:
13             self.hours = 0
14
15     def increment_minutes(self):
16         self.minutes += 1
17         if self.minutes == 60:
18             self.minutes = 0
19             self.increment_hours()
20
21 my_time = MyTime(23, 55)
22 for i in range(10):
23     my_time.increment_minutes()
24     my_time.print_time()
```

A full example of `MyTime` class. The example includes:

- Keyword `class` telling Python that what comes in the indented block is a definition of a class.
- `__init__` method which gets called (invoked) every time an instance of the class is created. This happens in line 21 of the code.
- `print_time` method used for printing the state of the instance, but leaving the instance unchanged.
- `increment_hours` which changes the state of the instance.
- `increment_minutes` which changes the state of the instance – notice that it uses `increment_hours` method.

The example does not include other possibilities (try it yourself):

- A method takes additional input, not only `self`. For example, implement a method with increments minutes for a certain number of minutes.
- A method with returns something. For example, implement a method with returns a string `'Time is <hh>:<mm>'`, where `<hh>` and `<mm>` are hours and minutes.