Chapter 8: Introduction to classes

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Oct 18, 2021

0.1 Plan for Week 42

Monday 18/10:

- Exercises 5.16, 5.18 (numpy, plotting, file reading, modules)
- Exercise 6.7, 6.9 (Dictionaries, strings)
- Introduction to classes

Thursday 21/10:

- Exercise 7.1, 7.2, 6.11
- More about classes
  - Protected attributes
  - Special methods (?)

0.2 Basics of classes (1)

- Classes are an essential part of *object oriented programming*
- We have used classes since day 1 in IN1900:

```
>>> S = "This is a string"
>>> type(S)
<class 'str'>
>>> L = S.split()
>>> type(L)
<class 'list'>
```

0.3 Basics of classes (2)

- Classes pack together data and functions that naturally belong together
- Every time we make a string object in Python, we create an *instance* of the built-in class `str`
- Calls like `S.split()` calls the function `split()` belonging the instance `S`
- We will now learn how to make our own classes

0.4 New terms and definition

- Class: Definition of a new data type, containing data and functions that naturally belong together.
- Instance: An object created from a class. After defining a class, we can create many instances of it.
- Method: A function bound to an instance of a class.
- Attribute: A variable bound to an instance of a class.
- A class consists of *attributes* and *methods*

0.5 Class = functions + data (variables) in one unit

- A class packs together data (attributes) and functions (methods) as *one single unit*
- As a programmer you can create a new class and thereby a new object type (like `float`, `list`, `file`, ...)
- A class is much like a module: a collection of "global" variables and functions that belong together
- There is only one instance of a module while a class can have many instances (copies)
- Modern programming applies classes to a large extent
- It usually takes some time to master the class concept
- Let's start with an example

0.6 Representing a function by a class; background (1)

Consider a function of t with a parameter v_0:

$$y(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

We need both v_0 and t to evaluate y (and $g = 9.81$), but how should we implement this?

Having t and v_0 as arguments?

```
def y(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

0.7 Representing a function by a class; background (2)

The implementation with two arguments usually works, but is not always convenient.

And what if the function is to be passed as an argument to another function that expects it to take a single argument only?

This is a very common situation in Python programs, consider for instance the implementation of Newton's method in Section 4.6 of the course book. The function `Newton(f, dfdx, x0, tol)` expects the first argument to be a Python function taking a single argument as input. If we pass it our `y(t,v0)` function it will fail.

0.8 Possible (sub-optimal) solutions

Having t as argument and v_0 as global variable?

```
def y(t):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

Motivation: $y(t)$ is a function of t only

Having t as argument and v_0 as local variable?

```
def y(t):
    v0 = 3
    g = 9.81
    return v0*t - 0.5*g*t**2
```

0.9 Representing a function by a class; idea

- Recall that a class packs together variables and functions that naturally belong together

- We can make a class holding `v0` and `g` as variables (attributes), and a function (method) `y(t)`
- With a class, `y(t)` can be a function of `t` only, but still have `v0` and `g` as parameters with given values.
- The class packs together a function `y(t)` and data (`v0`, `g`)

0.10 Representing a function by a class; technical overview

- We make a class `Y` for $y(t; v_0)$ with variables `v0` and `g` and a function `value(t)` for computing $y(t; v_0)$
- Any class should also have a function `__init__` for initialization of the variables

0.11 Representing a function by a class; the code

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Usage:

```
y = Y(v0=3)             # create instance (object)
v = y.value(0.1)        # compute function value
```

0.12 Representing a function by a class; the constructor

When we write

```
y = Y(v0=3)
```

we create a new variable (instance) `y` of type `Y`. `Y(3)` is a call to the *constructor*:

```
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81
```

0.13 What is this `self` variable? Stay cool - it will be understood later as you get used to it

- Think of `self` as `y`, i.e., the new variable to be created. `self.v0 = ...` means that we attach a variable `v0` to `self` (i.e., to `y`).
- `Y(3)` means `Y.__init__(y, 3)`, i.e., call the constructor with `self=y`, `v0=3`
- Remember: `self` is always the first parameter in the implementation of a method, but is never inserted in the call!
- After `y = Y(3)`, `y` has two variables `v0` and `g`

```
print(y.v0)
print(y.g)
```

> *In mathematics you don't understand things. You just get used to them.* John von Neumann, mathematician, 1903-1957.

0.14 Representing a function by a class; the value method

Here is the `value` method:

```
def value(self, t):
    return self.v0*t - 0.5*self.g*t**2
```

Example on a call:

```
v = y.value(t=0.1)
```

`self` is left out in the call, but Python automatically inserts `y` as the `self` argument inside the `value` method. Think of the call as

```
Y.value(y, t=0.1)
```

Inside `value` things "appear" as

```
return y.v0*t - 0.5*y.g*t**2
```

`self` gives access to "global variables" in the class object.

0.15 Recap; sketch of a general Python class

```
class MyClass:
    def __init__(self, p1, p2):
        self.attr1 = p1
        self.attr2 = p2

    def method1(self, arg):
        return self.attr1 + self.attr2 + arg

    def method2(self):
        print('Hello!')

m = MyClass(4, 10)
print(m.method1(-2))
m.method2()
```

0.16 Classes introduction - summary

- A class is simply a collection of functions and data that naturally belong together
- Functions in a class are usually called methods, data are called attributes
- We create instances (or objects) of a class, and each instance can have different values for the attributes
- All classes should have a method `__init__`, called a constructur, which is called every time a new instance is created
- The constructur will typically initialize all data in an instance
- All methods in a class should have `self` as first argument in the definition, but not in the call. This may be confusing at first, but one gets used to it.

0.17 Why use classes (1)?

- For short, simple Python programs, classes are never really necessary, but they can make a program more tidy and readable
- For large and complex programs, tidy and readable code is extremely important
- More important in other programming languages (Java, C++, etc)
- Python has convenient built-in data types (lists, dictionaries) that makes it less important to make your own classes

- Classes and object-oriented programming (OOP) are standard tools in software development
- OOP was invented at the University of Oslo (!)

0.18 Why use classes (2)

Think about how we have used the `str` class:

```
>>> a = "this is a string"
>>> type(a)
<class 'str'>
>>> l = a.split()
```

The Python developers could have solved this without classes, by making split a global function:

```
>>> a = "this is a string"
>>> l = split(a)
```

(Warning: this does not work, it is just a thought-example.) The advantage of the class solution is that it packs together data and functions that naturally belong together.

0.19 Recap; representing a function by a class

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Usage:

```
y = Y(v0=3)             # create instance (object)
v = y.value(0.1)        # compute function value
```

0.20 Representing a function by a class; summary

- Class `Y` collects the attributes `v0` and `g` and the method `value` as one unit
- `value(t)` is function of `t` only, but has automatically access to the parameters `v0` and `g` as `self.v0` and `self.g`
- Advantage: we can send `y.value` as an ordinary function of `t` to any other function that expects a function `f(t)` of one variable

```
def make_table(f, tstop, n):
    for t in linspace(0, tstop, n):
        print(t, f(t))

def g(t):
    return sin(t)*exp(-t)

make_table(g, 2*pi, 101)          # send ordinary function

y = Y(6.5)
make_table(y.value, 2*pi, 101)    # send class method
```

0.21 But what is this *self* variable?

More details are found in Section 8.1 (page 119) in the book

The syntax

```
y = Y(3)
```

can be thought of as

```
Y.__init__(y, 3)    # class prefix Y. is like a module prefix
```

0.22 How `self` works in the `value` method

```
v = y.value(2)
```

can alternatively be written as

```
v = Y.value(y, 2)
```

So, when we do `y.value(2)`, this is automatically translated to the call `Y.value(y,2)`.

0.23 Another class example: a bank account

- Attributes: name of owner, account number, balance
- Methods: deposit, withdraw, pretty print

```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

    def deposit(self, amount):
```

```
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def dump(self):
        s = f'{self.name}, {self.no}, balance: {self.balance}'
        print(s)
```

0.24 Example on using class Account

```
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson',  '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print("a1's balance:", a1.balance)
a1's balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```

0.25 How the class should not be used

Possible, but not intended use:

```
>>> a1.name = 'Some other name'
>>> a1.balance = 100000
>>> a1.no = '19371564768'
```

The assumptions on correct usage:

- The attributes should *not* be changed!
- The `balance` attribute can be viewed
- Changing `balance` is done through `withdraw` or `deposit`

Remedy: Attributes and methods not intended for use outside the class can be marked as *protected* by prefixing the name with an underscore (e.g., `_name`). This is just a convention - and no technical way of avoiding attributes and methods to be accessed.

0.26 Improved class with attribute protection (underscore)

```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):    # NEW - read balance value
        return self._balance

    def dump(self):
        s = f'{self.name}, {self.no}, balance: {self.balance}'
        print(s)
```

0.27 Usage of improved class AccountP

```
a1 = AccountP('John Olsson', '19371554951', 20000)
a1.withdraw(4000)

print(a1._balance)     # it works, but a convention is broken

print(a1.get_balance()) # correct way of viewing the balance

a1._no = '19371554955' # also works, but is a "serious crime"!
```

0.28 Question: Why is this useful?

Hint: Think of large library codes, that will be used by many other programmers for many years.