Ch.7: Introduction to classes (part 2)

Joakim Sundnes[1,2] Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Oct 27, 2017

Plan for Oct 27

- Recap of class introduction
- More class examples:
  - Bank account
  - A linear function
  - A circle
- Exercises (7.1), 7.2, 7.3, 7.10
- (More on classes; special methods)

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 (!)

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.

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
```

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`
- The great 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
```

Representing a function by a class; the general case

Given a function with $n+1$ parameters and one independent variable,

$$f(x; p_0, \ldots, p_n)$$

it is wise to represent `f` by a class where $p_0, \ldots, p_n$ are attributes and where there is a method, say `value(self, x)`, for computing $f(x)$

```
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn

    def value(self, x):
        return ...
```

Rough sketch of a general Python class

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

    def method1(self, arg):
        # can init new attribute outside constructor:
        self.attr3 = arg
        return self.attr1 + self.attr2 + self.attr3

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

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

It is common to have a constructor where attributes are initialized, but this is not a requirement - attributes can be defined whenever desired

Rough sketch of a general Python class

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

    def method1(self, arg):
        # can init new attribute outside constructor:
        self.attr3 = arg
        return self.attr1 + self.attr2 + self.attr3

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

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

It is common to have a constructor where attributes are initialized, but this is not a requirement - attributes can be defined whenever desired

Rough sketch of a general Python class

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

    def method1(self, arg):
        # can init new attribute outside constructor:
        self.attr3 = arg
        return self.attr1 + self.attr2 + self.attr3

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

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

It is common to have a constructor where attributes are initialized, but this is not a requirement - attributes can be defined whenever desired

But what is this self variable? I want to know now!

Warning

You have two choices:

1. follow the detailed explanations of what `self` really is (Section 7.1.3 in the book)
2. postpone understanding `self` until you have much more experience with class programming (suddenly `self` becomes clear!)

The syntax

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

can be thought of as

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

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)`.

Another class example: a bank account

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

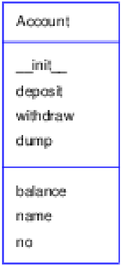
```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

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

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

    def dump(self):
        s = '%s, %s, balance: %s' % \
            (self.name, self.no, self.balance)
        print(s)
```

UML diagram of class Account



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
```

Use underscore in attribute names to avoid misuse

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.

Use underscore in attribute names to avoid misuse

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.

Use underscore in attribute names to avoid misuse

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.

Use underscore in attribute names to avoid misuse

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.

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 = '%s, %s, balance: %s' % \
            (self._name, self._no, self._balance)
        print(s)
```

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"!
```

Question: Why is this useful?

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

Another example: a class for a circle

- A circle is defined by its center point x_0, y_0 and its radius R
- These data can be attributes in a class
- Possible methods in the class: `area`, `circumference`
- The constructor initializes x_0, y_0 and R

```
class Circle:
    def __init__(self, x0, y0, R):
        self.x0, self.y0, self.R = x0, y0, R

    def area(self):
        return pi*self.R**2

    def circumference(self):
        return 2*pi*self.R
```

```
>>> c = Circle(2, -1, 5)
>>> print('A circle with radius %g at (%g, %g) has area %g' % \
...       (c.R, c.x0, c.y0, c.area()))
A circle with radius 5 at (2, -1) has area 78.5398
```

Class introduction - summary

- Classes pack together data and functions that naturally belong together
- We define a class, and then create *instances* (or objects) of that class
  - Different instances will have different data, but they all have the same functions operating on that data
- In IN1900 codes, classes are never really necessary, but sometimes convenient
- In "real-world" programs, with tens of 1000s of lines, the extra organization offered by classes may be the difference between a code that works and one that doesn't