

Introduction to Classes

Mojtaba Alaei

November 16, 2013

Content of the course

Class:

A class packs a set of data (variables) together with a set of functions operating on the data. The goal is to achieve more modular code by grouping data and functions into manageable (often small) units.

Problem: Functions with Parameters

In python we can define function with parameter by using `def`. For example for $y(t) = v_0 t - \frac{1}{2} g t^2$ or $g(x; A, a) = A e^{-ax}$:

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

Problem:

suppose we want to differentiate a function $f(x)$ at a point x , using the approximation:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (1)$$

So

```
def diff(f, x, h=1E-5):  
    return (f(x+h) - f(x))/h
```

Unfortunately, `diff` will not work with our `y(t, v0)` function. Calling `diff(y, t)` leads to an error inside the `diff` function, because it tries to call our `y` function with only one argument while the `y` function requires two.

Problem: Functions with Parameters

A Bad Solution: Global Variables.

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

So

```
v0 = 3  
dy = diff(y, 1)  
A = 1; a = 0.1  
dg = diff(g, 1.5)
```

The use of global variables is in general considered bad programming.

Solution: Representing a Function as a Class

A **class** contains a **set of variables (data)** and a **set of functions**, held together as one unit. The variables are visible in all the functions in the class. That is, we can view **the variables as "global"** in these functions.

Consider the function $y(t; v_0) = v_0 t - \frac{1}{2} g t^2$. We may say that v_0 and g , represented by the **variables v_0 and g** , constitute the **data**. A Python **function**, say **value(t)**, is needed to compute the value of $y(t; v_0)$ and this **function** must have **access** to the data v_0 and g , while t is an argument.

Solution: Representing a Function as a Class

So for this class we need the data v_0 and g , and the function $\text{value}(t)$, together as a class. In addition, a class usually has another function, called **constructor** for initializing the data. The constructor is always named `__init__`. Every class must have a name, often starting with a capital.

Implementation:

```
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 and Dissection

An object of a user-defined class (like `Y`) is usually called an instance. We need such an **instance** in order to use the data in the class and call the value function:

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

- Actually, `Y(3)` is **automatically** translated by Python to a call to the **constructor** `__init__` in class `Y`.
- The arguments in the call, here only the number 3, are always passed on as arguments to `__init__` after the `self` argument. That is, `v0` gets the value 3 and `self` is just dropped in the call.

This may be confusing, but it is a **rule** that the **self argument** is **never used in calls** to functions in classes.

With the instance `y`, we can compute the value `y(t = 0.1; v0 = 3)` by the statement:

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

With the instance y , we can compute the value $y(t = 0.1; v0 = 3)$ by the statement:

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

To access functions and variables in a class, we must prefix the function and variable names by the name of the instance and a dot: the value function is reached as `y.value`, and the variables are reached as `y.v0` and `y.g`.

For example, print the value of `v0` in the instance `y` by writing:

```
print y.v0
```

- We have already introduced the term "instance" for the object of a class.
- Functions in classes are commonly called **methods**,
- and variables (data) in classes are called **attributes**.

In our sample class Y we have two **methods**, `__init__` and `value`, and two **attributes**, `v0` and `g`.

The self Variable

Inside the constructor `__init__`, the argument `self` is a variable holding the new instance to be constructed. When we write:

```
self.v0 = v0  
self.g = 9.81
```

we define two new attributes in this instance. The `self` parameter is invisibly returned to the calling code. We can imagine that Python translates `y = Y(3)` to

```
Y.__init__(y, 3)
```

Let us look at a call to the `value` method to see a similar use of the `self` argument. When we write:

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

Python translates this to a call

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

such that the `self` argument in the `value` method becomes the `y` instance.

```
self.v0*t - 0.5*self.g*t**2
```

The self Variable

In the expression inside the value method,

```
self.v0*t - 0.5*self.g*t**2
```

`self` is `y` so this is the same as

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

The self Variable

The rules regarding "self" are listed below:

- Any class method must have self as first argument¹.
- self represents an (arbitrary) instance of the class.
- To access another class method or a class attribute, inside class methods, we must prefix with self, as in self.name, where name is the name of the attribute or the other method.
- self is dropped as argument in calls to class methods.

¹The name can be any valid variable name, but the name self is a widely established convention in Python.

Using Methods as Ordinary Functions

We may create several y functions with different values of v_0 :

```
y1 = Y(1)
y2 = Y(1.5)
y3 = Y(-3)
```

We can treat $y1.value$, $y2.value$, and $y3.value$ as ordinary Python functions of t , and then pass them on to any Python function that expects a function of one variable. In particular, we can send the functions to the `diff(f, x)` function:

```
dy1dt = diff(y1.value, 0.1)
dy2dt = diff(y2.value, 0.1)
dy3dt = diff(y3.value, 0.2)
```

A class can have a doc string, it is just the first string that appears right after the class headline. The convention is to enclose the doc string in triple double quotes `"""`:

```
class Y:  
    """The vertical motion of a ball."""  
    def __init__(self, v0):  
        ...
```

And to see the doc:

```
>>> Y.__doc__  
'The vertical motion of a ball.'
```

Making Classes Without the Class Construct

More Examples on Classes: Bank Accounts

Bank Accounts:

```
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
```

Usage:

```
>>> from classes import Account
>>> a1=Account('Mojtaba Alaei', '131031221', 100000)
>>> a1.withdraw(4000)
>>> a1.dump()
Mojtaba Alaei, 131031221, balance: 96000
>>> a1.name='Javad Alaei' # This is really a problem
>>> a1.dump()
Javad Alaei, 131031221, balance: 96000
```

More Examples on Classes: Bank Accounts

Other languages with class support usually have special keywords that can restrict access to class attributes and methods, but Python does not.

A special convention can be used: Any name starting with an underscore represents an attribute that should never be touched or a method that should never be called. One refers to names starting with an underscore as protected names:

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

More Examples on Classes: Bank Accounts

Here is class AccountP in action:

```
>>> a1 = AccountP('John Olsson', '19371554951', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a1.withdraw(3500)
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> print a1._balance # it works, but a convention is broken
13500
print a1.get_balance() # correct way of viewing the balance
13500
>>> a1._no = '19371554955' # this is a "serious crime"
```

More Examples on Classes: Phone Book

```
class Person:
    def __init__(self, name,
                 mobile_phone=None, office_phone=None,
                 private_phone=None, email=None):
        self.name = name
        self.mobile = mobile_phone
        self.office = office_phone
        self.private = private_phone
        self.email = email
    def add_mobile_phone(self, number):
        self.mobile = number
    def add_office_phone(self, number):
        self.office = number
    def add_private_phone(self, number):
        self.private = number
    def add_email(self, address):
        self.email = address
```

The object `None` is commonly used to indicate that a variable or attribute is defined, but yet not with a sensible value.

Usage:

```
>>> p1 = Person('Hans Hanson',
...             office_phone='767828283', email='h@hanshanson.com')
>>> p2 = Person('Ole Olsen', office_phone='767828292')
>>> p2.add_email('olsen@somemail.net')
>>> phone_book = [p1, p2]
```

More Examples on Classes: A Circle

More Examples on Classes: A Circle

```
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
```

More Examples on Classes: ASE

Special Methods

Some class methods have names starting and ending with a double underscore. These methods allow a special syntax in the program and are called special methods. The constructor `__init__` is one example. This method is automatically called when an instance is created (by calling the class as a function), **but we do not need to explicitly write `__init__`**.

The Call Special Method

If we could write just `y(t)`, instead of writing `y.value(t)`, the `y` instance would look as an ordinary function. Such a syntax is indeed possible and offered by the special method named `__call__`. Writing `y(t)` implies a call

```
y.__call__(t)
```

if class `Y` has the method `__call__` defined. We may easily add this special method:

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