Lecture 7

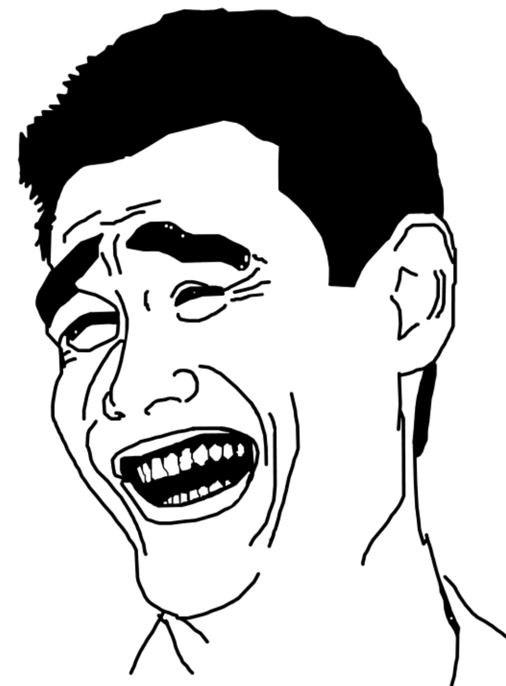
- Beginning OOP in Python

Qingyang "Young" Xu

April 21, 2015

What we've learned last week…

- NumPy arrays and functions
  - what is NumPy and why is it useful
  - slicing (accessing multiple entries)
  - some more weird stuff
- Plotting with matplotlib
- Basic SciPy curve-fitting
- Reading Python documentations



Review: NumPy array

- Given a 1D NumPy array, write the generic code which returns the unit vector.

```python
import numpy as np
a = np.array(list)


#approach 1: iterate
norm = 0
for i in range(a.size):
    norm += a[i] ** 2
a /= norm ** (1/2)
```

```python
from numpy import linalg
a = np.array(list)


#approach 2:  NumPy function
a /= linalg.norm(a)
```

Learning goals for today!

- After this lecture, you will…
- Understand OOP
  - what?
  - why?

- Write simple classes in Python…
  - what is the basic syntax?
  - what is duck typing?

What's OOP?

- Object Oriented Programming
- Before that, what is programming?
- A program (for our purposes)..

  - takes in some data (what you compute)

  - processes the data (how you compute)

  - returns new data (result of computation)

- In OOP, the fundamental element are objects. Data are organized into fields of objects and the processing of data into methods.

What is NOT OOP…

- How to model a Physics 91SI class?

```
def physics_91si(students, lecturer, lab_handout):
    sign_in_sheet = sign_in(students)
    present(lecturer)
    labs = do_lab(students, lab_handout)
    while !done(labs):
        q_and_a(students, lecturer, labs)
    return sign_in_sheet, labs
```

So what?

- This is an example of procedural programming.
- The most distinguishing feature of procedural programming is that most lines are in the form:

function(*arguments*)

how_to_compute(*what_to_compute*)

- Object oriented programming, in its simplest sense, reverses the above syntax into…

what_to_compute.*how_to_compute*()

object.method(*arguments*)

Why OOP?

- OOP makes it easier to model "real world".
- Reality: **thing** – **attribute** – **(inter)action**
- OOP: **object** – **field / attribute** – **methods**

- Suppose the system is made up of 10 balls (classical point of mass) on a billiard board.
- Each ball has its mass, position, and velocity.
- How to model the time evolution of the system?

Approach 1: Ramble Style

- Use brute force and basic tools to perform complicated and high-risk operations.
- In Procedural Programming, one would

  - make a list of mass

  - add another list of initial positions

  - add another list of initial velocities

  - keep track of the three lists all at once.
- But that's not how we think of the system in the real world…

Approach 2: OOP

- Instead of carpet bombing (with multiple lists), develop some **precision bombs** that are specifically tailored for this case…
- Establish the **class** of a "billiard ball" and use that as the fundamental **object** to model. Each ball is an **instance** of the class.
- Each billiard ball object memorizes its own attributes, or **fields**, and all instances of the same class perform the same actions, or **methods**.
- This allows us to write functions at the level of billiard ball, e.g. `def collision(ball1, ball2)`

So what?

- OOP tremendously simplifies the designing of any program, especially those which model the real-world scenarios.

	WHAT to model	**HOW to model**
What are they?	real billiard balls	**class** billiard ball
What do they have?	mass, position, velocity	**fields**
What do they do? What can be done to them?	move freely or under external force; bounce off walls	**methods**

- Now we are basically done, except we need to translate the above into Python.

```
class billiard_ball:
    # this is a special function called constructor
    # note we need a special argument self
    # no need to specify the fields before initializing
    def __init__(self, m, pos, vel):
        self.mass = m
        self.x_pos = pos[0]
        self.x_vel = vel[0]
        ...
 # constructs the billiard ball with the fields
ball1 = billiard_ball(.5, np.array([1, 2]), [0,1])
```

- Now write the method which models the linear motion of constant velocity of the ball within time interval **del_t**.

```
class billiard_ball:
    def __init__(self, m, pos, vel):
        ...
    def move(self, del_t):
        self.x_pos += self.x_vel * del_t
        self.y_pos += self.y_vel * del_t
    # similarly can model the motion
    # under sudden impulse
ball1.move(.1)
```

- Finally write the method which models the elastic collision of the ball onto the wall and incorporate that into the **move()** method.

```
class billiard_ball:
    def __init__(self, m, pos, vel):
    def bounce(self):
        if self.x_pos <= wall_left and  self.x_vel < 0:
            self.x_vel *= -1

    def move(self, del_t):
        ...
        # calls the bounce() function for each move
        self.bounce()
```

Special note: duck typing

```
class duck:
    def quack(self):
        print "Quack, quack!"
class person:
    def quack(self):
        print "I'm quacking like a duck!"
def forest_life(animal):
    animal.quack()
tom = duck()
jerry = person()
forest_life(tom)        # what is the output?
forest_life(jerry)      # should there be errors?
```

This is called duck typing…

- Since Python does not require the programmer to specify the type of variables, thus different classes (no inheritance) can have methods that share the same name but functions differently.
- Can you give an example of duck typing that we have seen before? (Hint: lecture 1)

```
x = 1
print x * 2
x = “Ha”
print x * 2
```

Testing your program:

- Due to the huge syntax freedom (e.g. duck typing) in Python, it is extremely easy to make semantic errors (i.e. syntactically correct code which does the wrong thing).
- In terms of testing and debugging, it is more useful for the code to crash and display the error message rather than to produce the incorrect output.
- Therefore, it is very useful to implement certain “time bombs” which are targeted against different semantic errors.

Approach 1: raise exception

- Exception in Python is analogous to Java and C++. There are several built-in exception types that are the most commonly called.

```
# implement duck-exclusive with exception
def duck_club(animal):
    if animal.__class__.__name__ != 'duck':
        raise TypeError("This is not a duck.")
    animal.quack()
```

Approach 2: use assert command

- The command name “assert” does not mean “state some fact with confidence”, but rather “check [condition] is satisfied and throw an error if otherwise”.

```
# implement the same function with assert
def duck_club(animal):
    assert animal.__class__.__name__ == ‘duck’, “This is not a duck!”
    # the above expression has to be in the same line
    animal.quack()
```

Syntax Summary: Class Definition

```
class billiard_ball:
    def __init__(self, m, pos, vel):  # constructor
        self.mass = m                 # initializes the fields
        ...
    def bounce(self):                 # note the "self" keyword
        if self.x_pos <= wall_left and  self.x_vel < 0:
            self.x_vel *= -1
        ...
    def move(self, del_t):
        self.x_pos += self.x_vel * del_t
        self.bounce()                 # calls another method
```

Looking forward…

- Next time, we'll learn…
  - advanced OOP: inheritance
  - testing program with exceptions
  - debugging with PDB
  - how to write documentation