Introduction to Scientific Python

Lecture 4: Modules, I/O, and Classes

Luke de Oliveira (lukedeo@stanford.edu)

October 8th, 2015

Contents

Importing a module

We can import a module by using `import`

E.g. `import math`

We can then access everything in `math`, for example the square root function, by:

```
math.sqrt(2)
```

Importing as

We can rename imported modules

E.g. `import math as m`

Now we can write `m.sqrt(2)`

In case we only need some part of a module

We can import only what we need using the `from ... import ...` syntax.

E.g. `from math import sqrt`

Now we can use `sqrt(2)` directly. This minimizes the load-time, nice for performance critical apps.

Import all from module

To import all functions, we can use `*`:

E.g. `from math import *`

Again, we can use `sqrt(2)` directly.

Note that this is considered bad practice! It makes it hard to understand where functions come from and what if several modules come with functions with same name.

Writing your own modules

It is perfectly fine to write and use your own modules. Simply import the name of the file you want to use as module.

E.g.

```
def helloworld():
    print 'hello, world!'

print 'this is my first module'
```

```
import firstmodule
firstmodule.helloworld()
```

What do you notice?

Only running code when main file

By default, Python executes all code in a module when we import it. However, we can make code run only when the file is the main file:

```
def helloworld():
    print 'hello, world!'

if __name__ == "__main__":
    print 'this only prints when run directly'
```

Try it! *Very* important when writing larger software projects.

Contents

File I/O

How to read from and write to disk.

The file object

- Interaction with the file system is pretty straightforward in Python.
- Done using *file objects*
- We can instantiate a file object using `open` or `file`

Opening a file

```
f = open(filename, option)
```

- filename: path and filename
- option:

'r'	read file
'w'	write to file
'a'	append to file

We need to close a file after we are done: `f.close()`

with open() as f

Very useful way to open, read/write and close file:

```
with open('data/text_file.txt', 'r') as f:
    print f.read()
```

Reading files

read()	Read entire line (or first n characters, if supplied)
readline()	Reads a single line per call
readlines()	Returns a list with lines (splits at newline)

Another fast option to read a file

```
with open('f.txt', 'r') as f:
    for line in f:
        print line
```

Reading files

read()	Read entire line (or first n characters, if supplied)
readline()	Reads a single line per call
readlines()	Returns a list with lines (splits at newline)

Another fast option to read a file

```
with open('f.txt', 'r') as f:
    for line in f:
        print line
```

Writing to file

Use `write()` to write to a file

```
with open(filename, 'w') as f:
    f.write("Hello, {}!\n".format(name))
```

More writing examples

```
# write elements of list to file
with open(filename, 'w') as f:
    for x in xs:
        f.write('{}\n'.format(x))

# write elements of dictionary to file
with open(filename, 'w') as f:
    for k, v in d.iteritems():
        f.write('{}: {}\n'.format(k, v))
```

Contents

Defining our own objects

So far, we have seen many objects in the course that come standard with Python.

- Integers
- Strings
- Lists
- Dictionaries
- etc

But often one wants to build (much) more complicated structures.

Defining our own objects

So far, we have seen many objects in the course that come standard with Python.

- Integers
- Strings
- Lists
- Dictionaries
- etc

But often one wants to build (much) more complicated structures.

Consider ‘building’ a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what’s in the room, what it’s made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle! Let’s think more about this...

Consider 'building' a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what's in the room, what it's made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle! Let's think more about this...

Consider ‘building’ a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what’s in the room, what it’s made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle! Let’s think more about this...

Consider ‘building’ a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what’s in the room, what it’s made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle! Let’s think more about this...

Object Oriented Programming

Construct our own objects

- House
- Room
- etc

The House has a `set` that holds the rooms, each room has a dict that maps location to furniture, each piece of furniture has a list of materials, etc, etc.

- Structure in familiar form
- Much easier to understand

Object Oriented Programming

Construct our own objects

- House
- Room
- etc

The House has a `set` that holds the rooms, each room has a dict that maps location to furniture, each piece of furniture has a list of materials, etc, etc.

- Structure in familiar form
- Much easier to understand

Object Oriented Programming

Express computation in terms of objects, which are instances of classes

Class	Blueprint (only one)
Object	Instance (many)

Classes specify attributes (data) and methods to interact with the attributes.

Object Oriented Programming

Express computation in terms of objects, which are instances of classes

Class	Blueprint (only one)
Object	Instance (many)

Classes specify attributes (data) and methods to interact with the attributes.

Python's way

In languages such as C++ and Java: data protection with private and public attributes and methods.

Not in Python: only basics such as inheritance.

Don't abuse power: works well in practice and leads to simple code.

Simplest example

```
# define class:
class Leaf:
    pass

# instantiate object
leaf = Leaf()

print leaf
# <__main__.Leaf instance at 0x10049df80>
```

Initializing an object

Define how a class is instantiated by defining the `__init__` *method*.

Seasoned programmer: in Python only one constructor method.

Initializing an object

The init or *constructor method*.

```
class Leaf:
    def __init__(self, color):
        self.color = color  # private attribute

redleaf = Leaf('red')
blueleaf = Leaf('blue')

print redleaf.color
# red
```

Note how we *access* object *attributes*.

Self

The `self` parameter seems strange at first sight.

It refers to the the object (instance) itself. If you come from C++, equivalent to `this`.

Hence `self.color = color` sets the color of the object `self.color` equal to the variable `color`.

Another example

Classes have *methods* (similar to functions)

```
class Stock():
    def __init__(self, name, symbol, prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

    def high_price(self):
        if len(self.prices) == 0:
            return 'MISSING PRICES'
        return max(self.prices)

apple = Stock('Apple', 'APPL', [500.43, 570.60])
print apple.high_price()
```

Recall: *list.append()* or *dict.items()*. These are simply class methods!

Another example

Classes have *methods* (similar to functions)

```
class Stock():
    def __init__(self, name, symbol, prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

    def high_price(self):
        if len(self.prices) == 0:
            return 'MISSING PRICES'
        return max(self.prices)

apple = Stock('Apple', 'APPL', [500.43, 570.60])
print apple.high_price()
```

Recall: *list.append()* or *dict.items()*. These are simply class methods!

Class attributes

```
class Leaf:
    n_leafs = 0  # class attribute: shared

    def __init__(self, color):
        self.color = color  # object attribute
        Leaf.n_leafs += 1

redleaf = Leaf('red')
blueleaf = Leaf('blue')

print redleaf.color
# red
print Leaf.n_leafs
# 2
```

Class attributes are shared among all objects of that class.

Class hierarchy through inheritance

It can be useful (especially in larger projects) to have a hierarchy of classes.

Example

- Animal
  - Bird
    - Hawk
    - Seagull
    - ...
  - Pet
    - Dog
    - Cat
    - ...
  - ...

Inheritance

Suppose we first define an abstract class

```
class Animal:
    def __init__(self, n_legs, color):
        self.n_legs = n_legs
        self.color = color

    def make_noise(self):
        print 'noise'
```

Inheritance

We can define sub classes and inherit from another class.

```
class Dog(Animal):
    def __init__(self, color, name):
        Animal.__init__(self, 4, color)
        self.name = name
    def make_noise(self):
        print self.name + ': ' + 'woof'

bird = Animal(2, 'white')
bird.make_noise()
# noise
brutus = Dog('black', 'Brutus')
brutus.make_noise()
# Brutus: woof
shelly = Dog('white', 'Shelly')
shelly.make_noise()
# Shelly: woof
```

Base methods

Some methods to override

- `__init__`: Constructor
- `__repr__`: Represent the object (machine)
- `__str__`: Represent the object (human)
- `__cmp__`: Compare

Example

Implementing Rational numbers

```python
class Rational:
    pass
```

Setup

What information should the class hold?

- Numerator
- Denominator

Setup

What information should the class hold?

- Numerator
- Denominator

Init

Implement the `__init__` method

```python
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Init

Implement the `__init__` method

```
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Issues

Issues?

```
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Ignore the division by 0 for now, more on that later.

Issues

Issues?

```
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Ignore the division by 0 for now, more on that later.

Greatest common divisor

$\frac{10}{20}$ and $\frac{1}{2}$ are the same rational.

Implement a `gcd(a, b)` function that computes the greatest common divisor of a and b.

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a%b)
```

Exercise: Verify Euclidean Algorithm

Greatest common divisor

```
class Rational:
    def __init__(self, p, q=1):
        g = gcd(p, q)
        self.p = p / g
        self.q = q / g
```

Why is this awesome?

Representing your class: Operator overloading

Implement `__repr__` or `__str__` early to `print`

Debugging

Operator overloading: adding two Rationals

Add Rationals just like Ints and Doubles?

`Rational(10,2) + Rational(4,3)`

To use +, we implement the `__add__` method

```
class Rational:
    # ...
    def __add__(self, other):
        p = self.p * other.q + other.p * self.q
        q = self.q * other.q
        return Rational(p, q)
    # ...
```

Operator overloading: Comparing

`__cmp__` compares objects

- If `self` is smaller than `other`, return a negative value
- If `self` and `other` are equal, return 0
- If `self` is larger than `other`, return a positive value

More on Operator Overloading

To learn more:

Google ‘Python operator overloading’.