CME 193: Introduction to Scientific Python

Lecture 6: Classes and iterators

Sven Schmit

stanford.edu/~schmit/cme193

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 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 more complicated structures.

From brick and mortar to houses

So far, I have taught you about brick and mortar, now it's time to learn how to build a house.

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!

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!

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!

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!

Object Oriented Programming

Construct our own objects

- House
- Room
- etc

- Structure in familiar form
- Much easier to understand

Object Oriented Programming

Construct our own objects

- House
- Room
- 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, classes provide data protection with private and public attributes and methods.

Not the case in Python: only basics such as inheritance.

Up to programmers not to 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

We can define how a class is instantiated by defining the `__init__` *method*.

For the more seasoned programmer: in Python there can only be one constructor method.

Initializing an object

The init or *constructor method*.

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

    def __init__(self, color):
        self.color = color  # private variable
        Leaf.n_leafs += 1

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

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

Note how we *access* object *attributes*.

Self

The `self` parameter seems strange at first sight.

It refers to the the object (instance) itself.

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!

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
```

Why inheritance

Often very useful:

An abstract class that implements general functionality can be combined with several subclasses that implement details.

Example: statistical package with a general *Model* class. Subclasses can include *Linear regression*, *Logistic regression*, etc.

Why inheritance

Often very useful:

An abstract class that implements general functionality can be combined with several subclasses that implement details.

Example: statistical package with a general *Model* class. Subclasses can include *Linear regression*, *Logistic regression*, etc.

Base methods

Base methods which you can override

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

Compare

The `__cmp__` method should be implemented as follows:

- 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 useful methods

Some more on this later!

- `__contains__` for the `in` keyword
- `__iter__` and `next` for iterators

Exercise

One of the exercises today is to implement a class for rational numbers, i.e. fractions.

Let's start together

Setup

What information should the class hold?

- Numerator
- Denominator

Setup

What information should the class hold?

- Numerator
- Denominator

Init

Implement the `__init__` method

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

Init

Implement the `__init__` method

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

Issues

What are the issues with the code?

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

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

Issues

What are the issues with the code?

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

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

Greatest common divisor

We really would like to ensure that $\frac{10}{20}$ and $\frac{1}{2}$ are the same number.

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

Hint: Google Euclidean algorithm

Greatest common divisor

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

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

Greatest common divisor

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

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

Add two rational numbers

Implement that `__add__` method to add two rational numbers as in

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

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

Add two rational numbers

Implement that `__add__` method to add two rational numbers as in

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

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

Contents

- Classes
- Generators and Iterators
- Exercises

For loops

Recall that in Python we can loop over the elements in a list simply by saying

```
for elem in li:
    # do something
```

We also say that we are iterating over the list.

Fibonnaci numbers

Say we want to iterate over the first n Fibonacci numbers:

```
for elem in fib(n):
    # do something
```

How to implement Fib(n)?

fib(n)

We can use our function from the *lists* homework, which returns a list with the first n Fibonacci numbers.

What happens when n is large, say $n = 10^6$ or more generally $n = 10^k$?

Problem: we have to store the entire list...

Eventually we will run out of memory.

fib(n)

We can use our function from the *lists* homework, which returns a list with the first n Fibonacci numbers.

What happens when n is large, say $n = 10^6$ or more generally $n = 10^k$?

Problem: we have to store the entire list...

Eventually we will run out of memory.

fib(n)

We can use our function from the *lists* homework, which returns a list with the first n Fibonacci numbers.

What happens when n is large, say $n = 10^6$ or more generally $n = 10^k$?

Problem: we have to store the entire list...

Eventually we will run out of memory.

Generators

Instead, we can use so called generators

Only difference with functions: use `yield` instead of `return`

Example: Range

```
def myrange(end, start=0, step=1):
    if start > end:
        start, end = end, start
    current = start
    yield current
    while current+step < end:
        current += step
        yield current
```

Advantage over lists

Note, in this case we never store the entire list.

This scales much better!

(though Fibonacci numbers scale poorly, but you get the idea)

Back to Fibonacci

How to write a generator for the Fibonacci sequence?

```
def fib(n):
    a, b = 0, 1
    i = 1
    while i < n:
        a, b = b, a + b
        i += 1
        yield a

for elem in fib(10):
    print elem
```

Generators are a simple form of iterators

Iterators underlie many aspects of Python such as

- List comprehensions
- Generators

Iterators are classes

- Init method to initialize iterator
- `__iter__` method to set up the iterator
- `next` method to loop over the entries

More control, but also more code.

Range as iterator

A slight overkill...

```
class Range:
    def __init__(self, end, start=0, step=1):
        if end < start:
            end, start = start, end
        self.current = start
        self.end = end
        self.step = step

    def __iter__(self):
        return self

    def next(self):
        c = self.current
        if c >= self.end:
            raise StopIteration
        self.current += self.step
        return c
```

Fibonacci iterator

```
class Fib:
    def __init__(self, n):
        self.n = n
        self.i = 0

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def next(self):
        if self.i >= self.n: raise StopIteration
        self.i += 1
        self.a, self.b = self.b, self.a + self.b
        return self.a
```

Contents

Exercises

Next lecture: Eli Bressert