Introduction to Scientific Python

Lecture 3: Lists, Data Structures, and Modules

Luke de Oliveira (lukedeo@stanford.edu)

October 6th, 2015

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

Some remarks about the final project

- You learn Python by writing Python
- I want to give you freedom to do what you like
- People from very different backgrounds and programming experience
- Do not worry too much about (failing) the project
- Just spend some time outside of class writing Python
- Use your own judgement
- Learn to chain together some concepts

Proposal

- 1-2 paragraph pdf outlining your project ideas
- Email me (lukedeo@stanford.edu) with the subject line:

  `[CME193 PROJECT]LastName-FirstName`.

  I have a filter set up for these.
- Due this Thursday before class (don't stress, this is fun!)

Today

- I went a bit too slow last time, we'll speed up.
- Lists (continued), tuples, dictionaries, sets, string, modules
- How are people feeling?
- I want you to learn and have fun!

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

More control over lists

- `len(xs)`
- `xs.append(x)`
- `xs.count(x)`
- `xs.insert(i, x)`
- `xs.sort()` and `sorted(xs)`: what's the difference?
- `xs.remove(x)`
- `xs.pop()` or `xs.pop(i)`
- `x in xs`

All these can be found in the Python documentation, google: 'python list'

Looping over elements

It is very easy to loop over elements of a list using `for`, we have seen this before using `range`.

```
someIntegers = [1, 3, 10]

for integer in someIntegers:
    print integer,
# 1 3 10

# What happens here?
for integer in someIntegers:
    integer = integer*2
```

Looping over elements

Using `enumerate`, we can loop over both element and index at the same time.

```
myList = [1, 2, 4]

for index, elem in enumerate(myList):
    print '{0}) {1}'.format(index, elem)

# 0) 1
# 1) 2
# 2) 4
```

Map

We can apply a function to all elements of a list using `map`

```
l = range(4)
print map(lambda x: x*x*x, l)
# [0, 1, 8, 27]
```

Filter

We can also filter elements of a list using `filter`

```
l = range(8)

print filter(lambda x: x % 2 == 0, l)
# [0, 2, 4, 6]
```

List comprehensions

A very powerful and concise way to create lists is using *list comprehensions*

```
print [i**2 for i in range(5)]
# [0, 1, 4, 9, 16]
```

This is often more readable than using `map` or `filter`

List comprehensions

```
ints = [1, 3, 10]

[i * 2 for i in ints]
# [2, 6, 20]

[[i, j] for i in ints for j in ints if i != j]
# [[1, 3], [1, 10], [3, 1], [3, 10], [10, 1], [10, 3]]

[(x, y) for x in xrange(3) for y in xrange(x+1)]
# ...
```

Note how we can have a lists as elements of a list!

List comprehensions

```
ints = [1, 3, 10]

[i * 2 for i in ints]
# [2, 6, 20]

[[i, j] for i in ints for j in ints if i != j]
# [[1, 3], [1, 10], [3, 1], [3, 10], [10, 1], [10, 3]]

[(x, y) for x in xrange(3) for y in xrange(x+1)]
# ...
```

Note how we can have a lists as elements of a list!

Implementing map using list comprehensions

Let's implement `map` using list comprehensions

```
def my_map(f, xs):
    return [f(x) for x in xs]
```

Implement filter by yourself in one of the exercises.

Implementing map using list comprehensions

Let's implement `map` using list comprehensions

```
def my_map(f, xs):
    return [f(x) for x in xs]
```

Implement filter by yourself in one of the exercises.

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

Tuples

Seemingly similar to lists

```
>>> myTuple = (1, 2, 3)
>>> myTuple[1]
2
>>> myTuple[1:3]
(2, 3)
```

Tuples are immutable

Unlike lists, we cannot change elements.

```
>>> myTuple = ([1, 2], [2, 3])
>>> myTuple[0] = [3,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support item assignment
>>> myTuple[0][1] = 3
>>> myTuple
([1, 3], [2, 3])
```

Packing and unpacking

```
t = 1, 2, 3
x, y, z = t

print t  # (1, 2, 3)
print y  # 2
```

Functions with multiple return values

```
def simple_function():
    return 0, 1, 2

print simple_function()
# (0, 1, 2)
```

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

Dictionaries

A dictionary is a *collection* of *key-value* pairs.

An example: the keys are all words in the English language, and their corresponding values are the meanings.

Lists + Dictionaries = $$$

Defining a dictionary

```
>>> d = {}
>>> d[1] = "one"
>>> d[2] = "two"
>>> d
{1: 'one', 2: 'two'}
>>> e = {1: 'one', 'hello': True}
>>> e
{1: 'one', 'hello': True}
```

Note how we can add more key-value pairs at any time. Also, only condition on keys is that they are *immutable*.

No duplicate keys

Old value gets overwritten instead!

```
>>> d = {1: 'one', 2: 'two'}
>>> d[1] = 'three'
>>> d
{1: 'three', 2: 'two'}
```

Access

We can access values by keys, but not the other way around

```
>>> d = {1: 'one', 2: 'two'}
>>> print d[1]
```

Furthermore, we can check whether a key is in the dictionary by

`key in dict`

Access

We can access values by keys, but not the other way around

```
>>> d = {1: 'one', 2: 'two'}
>>> print d[1]
```

Furthermore, we can check whether a key is in the dictionary by

`key in dict`

All keys, values or both

Use `d.keys()`, `d.values()` and `d.items()`

```
>>> d = {1: 'one', 2: 'two', 3: 'three'}
>>> d
{1: 'one', 2: 'two', 3: 'three'}
>>> d.keys()
[1, 2, 3]
>>> d.values()
['one', 'two', 'three']
>>> d.items()
[(1, 'one'), (2, 'two'), (3, 'three')]
```

So how can you loop over dictionaries?

Small exercise

Print all key-value pairs of a dictionary

```
>>> d = {1: 'one', 2: 'two', 3: 'three'}
>>> for key, value in d.items():
...     print key, value
...
1 one
2 two
3 three
```

Instead of `d.items()`, you can use `d.iteritems()` as well. Better performance for large dictionaries.

Small exercise

Print all key-value pairs of a dictionary

```
>>> d = {1: 'one', 2: 'two', 3: 'three'}
>>> for key, value in d.items():
...     print key, value
...
1 one
2 two
3 three
```

Instead of `d.items()`, you can use `d.iteritems()` as well. Better performance for large dictionaries.

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

Sets

Sets are an unordered collection of unique elements. Think math!

```
>>> basket = ['apple', 'orange', 'apple',
    'pear', 'orange', 'banana']
>>> fruit = set(basket)  # create a set
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit  # fast membership testing
True
>>> 'crabgrass' in fruit
False
```

Implementation: like a dictionary only keys.

from: Python documentation

Set comprehensions

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

from: Python documentation

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

Strings

Let's quickly go over *strings*. This is very useful!

- Strings hold a sequence of characters.
- Strings are *immutable*
- We can slice strings just like lists and tuples
- Between quotes or triple quotes

Everything can be turned into a string!

We can turn anything in Python into a string using `str`.

This includes dictionaries, lists, tuples, etc. Can make your own! `__str__()` member of class

String formatting

- Special characters: \n, \t, \b, etc
- Add variables: %s, %f, %e, %g, %d, or use `format`
- If you're scraping web, docs, etc., you might have *trailing newlines*. Use `.strip()`!
- Use `.replace('old', 'new')` to replace all instances of 'old' with 'new' in a string.

```
fl = 0.23
wo = 'Hello'
inte = 12

print "s: {} \t f: {:0.1f} \n i: {}".format(wo, fl, inte)
# s: Hello     f: 0.2
#  i: 12
```

Split

To split a string, for example, into separate words, we can use `split()`

```
text = 'Hello, world!\n How are you?'
text.split()
# ['Hello,', 'world!', 'How', 'are', 'you?']
```

Split

What if we have a comma seperated file with numbers seperated by commas?

```
numbers = '1, 3, 2, 5'
numbers.split()
# ['1,', '3,', '2,', '5']

numbers.split(', ')
# ['1', '3', '2', '5']

[int(i) for i in numbers.split(', ')]
# [1, 3, 2, 5]
```

Use the optional argument in `split()` to use a custom seperator.

What to use for a tab seperated file?

Split

What if we have a comma seperated file with numbers seperated by commas?

```
numbers = '1, 3, 2, 5'
numbers.split()
# ['1,', '3,', '2,', '5']

numbers.split(', ')
# ['1', '3', '2', '5']

[int(i) for i in numbers.split(', ')]
# [1, 3, 2, 5]
```

Use the optional argument in `split()` to use a custom seperator.

What to use for a tab seperated file?

UPPER and lowercase

There are a bunch of useful string functions, such as `.lower()` and `.upper()` that turn your string in lower- and uppercase.

Note: To quickly find all functions for a string, we can use `dir`

```
text = 'hello'
dir(text)
```

join

Another handy function: `join`.

We can use join to create a string from a list.

```
words = ['hello', 'world']
' '.join(words)

''.join(words)
# 'helloworld'

' '.join(words)
# 'hello world'

', '.join(words)
# 'hello, world'
```

Contents

- Project and Comments
- Lists
- Tuples
- Dictionaries
- Sets
- Strings
- Modules

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.