Python Tutorial

CS/CME/BioE/Biophys/BMI 279
Oct. 17, 2017
Rishi Bedi
Outline

• Python2 vs Python3
• Python syntax
• Data structures
• Functions
• Debugging
• Classes
• The NumPy Library

Many examples adapted from Sam Redmond’s CS41
Python2 vs Python3

• Our assignments are written for **Python2**
• You are welcome to use Python3, but you’re on your own if you run into bugs; and will likely have to make some modifications to get things working
• Python2 will be maintained until **2020**

http://py3readiness.org/
Python Syntax
# Variables

<table>
<thead>
<tr>
<th>Python</th>
<th>Java/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 5</td>
<td>int x = 5;</td>
</tr>
<tr>
<td>y = x + 7</td>
<td>int y = x + 7;</td>
</tr>
<tr>
<td>z = 3.14</td>
<td>double z = 3.14;</td>
</tr>
<tr>
<td>name = “Rishi”</td>
<td>String name = “Rishi”; // Java</td>
</tr>
<tr>
<td></td>
<td>string name(“Rishi”); // C++</td>
</tr>
<tr>
<td>1 == 1 # =&gt; True</td>
<td>1 == 1 # =&gt; true</td>
</tr>
<tr>
<td>5 &gt; 10 # =&gt; False</td>
<td>5 &gt; 10 # =&gt; false</td>
</tr>
<tr>
<td>True and False # =&gt; False</td>
<td>true &amp;&amp; false # =&gt; false</td>
</tr>
<tr>
<td>not False # =&gt; True</td>
<td>!(false) # =&gt; true</td>
</tr>
</tbody>
</table>

Variables are not statically typed!
Data Structures

- Lists
- Tuples
- Dictionaries
- Iteration
- List Comprehensions
Lists

easy_as = [1, 2, 3]

Square brackets delimit lists

Commas separate elements
Lists

# Create a new list
empty = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]

# Lists can contain elements of different types
mixed = [4, 5, "seconds"]

# Append elements to the end of a list
numbers.append(7)  # numbers == [2, 3, 5, 7]
numbers.append(11)  # numbers == [2, 3, 5, 7, 11]
Lists

# Access elements at a particular index
numbers[0]  # => 2
numbers[-1]  # => 11

# You can also slice lists - the same rules apply
letters[:3]  # => ['a', 'b', 'c']
numbers[1:-1]  # => [3, 5, 7]

# Lists really can contain anything - even other lists!
x = [letters, numbers]
x  # => [['a', 'b', 'c', 'd'], [2, 3, 5, 7, 11]]
x[0]  # => ['a', 'b', 'c', 'd']
x[0][1]  # => 'b'
x[1][2:]  # => [5, 7, 11]
Tuples

Parentheses delimit tuples

Like lists, but immutable

my_tup[0] = 5 => Error!

Commas separate elements
Dictionaries

dict = {'a': 2,
        'b': 3}

Commas separate entries
Dictionaries

d = {"one": 1, "two": 2, "three": 3}

len(d.keys())  # => 3

print d['one']   # => 1
print d['five']  # => ERROR!

d['five'] = 5    # => OK, creates new key

d.keys()        # iterator over k
d.values()      # iterator over v
d.items()       # iterator over (k, v) pairs
Iteration

Most data structures can be iterated over in the same way:

```python
dict = {'a': 10, 'b': 15}
for key in dict:
    print key, dict[key]
```

When iterating over a dictionary like this, we iterate over the keys.

```python
mylist = ['a', 'b', 'c']
for item in mylist:
    print item
```

```python
mytuple = ('a', 'b', 'c')
for item in mytuple:
    print item
```

Note we don’t need the index of the element to access it.
Iteration

We can also iterate over indices:

```python
for i in range(4):
    print i,           # 0 1 2 3
for i in range(1, 10, 2):
    print i,         # 1 3 5 7 9
mylist = ['a', 'b', 'c']  # 0 'a'
for i in range(len(mylist)):
    print i, mylist[i]  # 1 'b'
mylist = ['a', 'b', 'c']  # 0 'a'
for idx, item in enumerate(mylist):
    print idx, item    # 1 'b'
# 2 'c'
mylist = ['a', 'b', 'c']  # 0 'a'
for idx, item in enumerate(mylist):
    print idx, item    # 1 'b'
# 2 'c'
```
Input: nums = [1, 2, 3, 4, 5]
Goal: sq_nums = [1, 4, 9, 16, 25]

Here's how we could already do this:

```python
sq_nums = []
for n in nums:
    sq_nums.append(n**2)
```

Or... we could use a comprehension:

```python
sq_nums = [n ** 2 for n in nums]
```

- square brackets show we're making a list
- apply some operation to the loop variable
- loop over the specified iterable
More List Comprehensions

Template:
```
new_list = [f(x) for x in iterable]
```

```
words = ['hello', 'this', 'is', 'python']
caps = [word.upper() for word in words]
```

```
powers = [(x**2, x**3, x**4) for x in range(10)]
```

Remember this doesn’t have to be a list! Can be any iterable.
def fn_name(param1, param2):
    value = do_something()
    return value

def isEven(num):
    return (num % 2 == 0)

myNum = 100
if isEven(myNum):
    print str(myNum) + " is even"

• def starts a function definition
• return is optional
  ◦ if either return or its value are omitted, implicitly returns None
• Parameters have no explicit types
Putting Functions and List Comprehensions Together

Goal: given a list of numbers, generate a list that contains True for every even number and False for every odd number

```
def isEven(num):
    return (num % 2 == 0)
```

Template:
```
new_list = [f(x) for x in iterable]
```

```
numbers = [5, 18, 7, 9, 2, 4, 0]
isEvens = [isEven(num) for num in numbers]
```

## isEvens = [False, True, False, False, True, True, True]
Importing Modules

```
from math import exp
from random import random
```

Can now do:
- `exp(0.5)` to compute $e^{0.5}$
- `random()` to generate uniform random number over [0,1]

But if we had imported the whole modules, like this...

```
import math
import random
```

Then we would call the functions like this:

```
math.exp(0.5)
random.random()
```
Debugging Tricks

• “print line debugging”
  – At various points in your code, insert print statements that log the state of the program

• You will probably want to print some strings with some variables
  – You could just join things together like this:
    ```python
    print 'Variable x is equal to ' + str(x)
    ```
  – … but that gets unwieldy pretty quickly
  – The `format` function is much nicer:
    ```python
    print 'x, y, z are equal to {}, {}, {}'.format(x, y, z)
    ```
Python Debugger (pdb)

• Python Debugger: pdb
  – insert the following in your program to set a breakpoint
  – when your code hits these lines, it’ll stop running and launch an interactive prompt for you to inspect variables, step through the program, etc.

```python
import pdb
pdb.set_trace()
```

n to step to the next line in the current function
s to step into a function
c to continue to the next breakpoint
you can also run any Python command, like in the interpreter
```python
class Predictor(object):
    def __init__(self, nIters, name):
        self.nIters = nIters
        self.name = name

    def predict(self, start):
        raise NotImplementedError("Predictor should not be instantiated")

class MonteCarloPredictor(Predictor):
    def predict(self, start):
        ## Do some stuff
        for x in self.nIters:
            # do some stuff
            pass

myPredictor = MonteCarloPredictor(nIters=1000, name='myPred')
myPredictor.predict(pose)
```

__init__ gets called when you instantiate an object

Need to refer to member variables with self. prefix inside the class

...but outside the class, refer to member variables/functions using the object's name
Numeric Computing using NumPy

- Python’s built-in datatypes are very flexible
- They aren’t optimized for fast numerical calculations, especially on large multi-dimensional matrices
- NumPy is a widely-used 3rd party package which adds such support to Python
- Sister library for scientific computing: SciPy
NumPy Example

```python
>>> import numpy as np
>>> mat = np.ones((3,3))
>>> print(mat)
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
>>> mat[1,1] = 5
>>> print(mat)
[[ 1.  1.  1.]
 [ 1.  5.  1.]
 [ 1.  1.  1.]]
>>> vec = np.array([1, 2, 3])
>>> np.dot(mat, vec)
array([  6.,  14.,   6.])
```

I can rename my module when I import it for convenience.

It looks a lot like a list of lists!

Create arrays using `np.array`

Support for various linear algebra operations like dot products.
NumPy Example

Are the absolute values of all elements in the array less than 5?

```python
>>> a = np.array([1, -2, 3])
>>> b = np.array([1, 2, -6])
>>> np.all(np.abs(a) < 5)
True
>>> np.all(np.abs(b) < 5)
False
```

Can apply operations like absolute value element-wise

np.all checks whether all elements evaluate to True