

INTRO to DATA SCIENCE

CLASS 2: INTRO to PYTHON

Rob Hall

DAT SF 19// November 2, 2015

RECAP

LAST TIME:

- WHAT IS A DATA SCIENTIST?**
- WHAT IS DATA SCIENCE?**
- DATA SCIENCE WORKFLOW(S)**

QUESTIONS?

AGENDA

I. INTRO TO PYTHON

II. PYTHON STRENGTHS & WEAKNESSES

III. PYTHON DATA STRUCTURES

IV. PYTHON CONTROL FLOW

INTRO TO PYTHON

I. INTRO TO PYTHON

Q: What is Python?

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

Q: What is Python?

A: An **open source**, high-level, dynamic scripting language.

- ***open source: free! (both binaries and source files)***

Q: What is Python?

A: An open source, **high-level**, dynamic scripting language.

- **open source:** *free! (both binaries and source files)*
- **high-level:** *interpreted (not compiled)*

Q: What is Python?

*A: An open source, high-level, **dynamic** scripting language.*

- **open source:** *free! (both binaries and source files)*
- **high-level:** *interpreted (not compiled)*
- **dynamic:** *things that would typically happen at compile time happen at runtime instead (eg, dynamic typing)*

```
>>> x = 1
>>> x
1
>>> x = 'horse'
'horse'
>>>
```

Q: What is Python?

*A: An open source, high-level, dynamic **scripting language**.*

- **open source:** *free! (both binaries and source files)*
- **high-level:** *interpreted (not compiled)*
- **dynamic:** *things that would typically happen at compile time happen at runtime instead (eg, dynamic typing)*
- **scripting language:** *“middle-weight”*

Why Python for Data Science?

Command Line Interface (CLI) – for even quicker prototyping

Straight- “sugar-free lite” syntax

Multiple programming paradigms

Large corpus of available libraries

Wide-use means extensive community support – stackoverflow, et al.

Python is an open source project which is maintained by a large and very active community.

It was originally created by Guido Van Rossum in the 1990s, who currently holds the title of Benevolent Dictator For Life (BDFL).



Guido...the early years



The presence of a BDFL means that Python has a unified design philosophy.

This design philosophy emphasizes readability and ease of use, and is codified in PEP8 (the Python style guide) and PEP20 (the Zen of Python).

NOTE: PEPs* are the public design specs that the language follows.

*Python Enhancement Proposals

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

PEP 20: THE ZEN OF PYTHON

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

II. PYTHON STRENGTHS & WEAKNESSES

Python's popularity comes from the strength of its design.

The syntax looks like pseudocode, and it is explicitly meant to be clear, compact, and easy to read.

*This is usually summarized by saying Python is an **expressive** language.*

Python is also an extremely versatile language, and it attracts fans from many different walks of life:

web development Django

<https://www.djangoproject.com/>

data analysis Pandas

<http://pandas.pydata.org/>

systems admin Fabfile

<http://docs.fabfile.org/en/1.8/>

Config mgmt SaltStack

<http://www.saltstack.com>

(etc) ...more!

<https://github.com/search?q=python>

Python supports multiple programming paradigms, such as:

- imperative programming*
- object oriented programming*
- functional programming (really function-esqe)*

```
print "Print some sequential numbers to the screen"  
for i in range(5) :  
    print i
```

```
>>> x = range(5)
>>> x
[0, 1, 2, 3, 4]
>>> [k**2 for k in x]
[0, 1, 4, 9, 16]
>>>
```

NOTE: This is called a *list comprehension*

*Another great strength is the **Python Standard Library**.*

This is a collection of packages that ships with the standard Python distribution, and “...covers everything from asynchronous processing to zip files”.

*The advantages of the PSL are usually described by saying that Python comes with **batteries included**.*

Ultimately, Python's most important strength is that it's easy to learn and easy to use.

Because there should be only one way to perform a given task, things frequently work the way you expect them to.

- paraphrased from PEP20 ("The Zen of Python")

Takeaway: This is a huge luxury!

Q: Python sounds amazing. What is it bad at?

For one thing, Python is slower than a lower-level language (but keep in mind that this is a conscious tradeoff).

*Many people would say that Python's Achilles heel is concurrency. This is a result of the **Global Interpreter Lock** (again, a conscious design decision).*

There are some other subtleties regarding dynamic typing that people occasionally dislike, but again this is intentional (and a matter of opinion).

III. PYTHON DATA STRUCTURES

*The most basic data structure is the **None** type. This is the equivalent of NULL in other languages.*

There are four basic numeric types:

1. **int** ($< 2^{63}$) / **long** ($\geq 2^{63}$)*

* on 64-bit OS X/Linux, `sys.maxint = 2**63-1`

2. **float** (a “decimal”)

3. **bool** (True/False) or (1/0)

4. **complex** (“imaginary”)

```
>>> type(None)
<type 'NoneType'>
>>> type(1)
<type 'int'>
>>> type(2.5)
<type 'float'>
>>> type(True)
<type 'bool'>
>>> type(2+3j)
<type 'complex'>
```

*The next basic data type is the array, implemented in Python as a **list**.
A list is a (zero-base numbered), ordered collection of elements, and these elements can be of arbitrary type.
Lists are **mutable**, meaning they can be changed in-place.*

```
>>> a = [1, 'b', True]
>>> a[2]
True
>>> a[1] = 'aa'
>>> a
[1, 'aa', True]
```

Tuples: *immutable arrays of arbitrary elements.*

```
>>> x = (1, 'a', 2.5)
>>> x[0]
1
>>> x[0]='b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a,b = (1,2)
>>> a
1
```

*Tuples are frequently used behind the scenes in a special type of variable assignment called **tuple packing/unpacking**.*

*The **string** type in Python represents an immutable ordered array of characters (note there is no char type).*

Strings support slicing and indexing operations like arrays, and have many other string-specific functions as well.

String processing is one area where Python excels.

*Associative arrays (or hash tables) are implemented in Python as the **dictionary** type. This is a very efficient and useful structure that Python's internal functions use extensively.*

```
>>> this_class={'subject':'Data Science','location':'501 Folsom',  
'duration':11,'has_begun':True}  
>>> this_class['subject']  
'Data Science'  
>>> this_class['has_begun']  
True
```

*Dictionaries are unordered collections of **key-value** pairs, and dictionary keys must be immutable.*

Sets are unordered mutable collections of distinct elements.

```
>>> y = set([1,1,2,3,5,8])  
>>> y  
set([8, 1, 2, 3, 5])
```

These are particularly useful for checking membership of an element and for ensuring element uniqueness.

*Our final example of a “data type” is the Python **file object**. This example represents an open connection to a file on your laptop.*

```
>>> with open('output_file.txt', 'w') as f:  
...     f.write('test')
```

*These are particularly easy to use in Python, especially using the **with** statement context manager, which automatically closes the file handle when it goes out of scope.*

IV. PYTHON CONTROL FLOW

*Python has a number of control flow tools that will be familiar from other languages. The first is the **if-else statement**, whose compound syntax looks like this:*

```
>>> x, y = False, False
>>> if x :
...     Print 'x is True'
... elif y :
...     Print 'y is True'
... else :
...     Print 'Neither...'
...
Neither...
```

A while loop executes while a given condition evaluates to True.

```
>>> x = 0
>>> while (x < 3) :
...     print 'HELLO!'
...     x += 1
...
HELLO!
HELLO!
HELLO!
```

The familiar (& useful) for loop construct executes a block of code for a range of values.

```
>>> for k in range(4) :  
...     print k**2  
...  
0  
1  
4  
9
```

The object that a for loop iterates over is called (appropriately) an iterable.

*A useful but possibly unfamiliar construct is the **try-except** block:*

```
>>> try:
...     print undefined_variable
... except :
...     print 'An Exception has been caught'
...
An Exception has been caught
```

*This is useful for catching and dealing with errors, also called **exception handling**.*

*Python allows you to define **custom** functions:*

```
>>> def x_minus_3(x) :  
...     return x - 3  
...  
>>> x_minus_3(12)  
9
```

*NOTE: Functions can optionally return a value with a **return** statement (as this example does).*

*Functions can take a number of **arguments** as inputs, and these arguments can be provided in two ways:*

*1) as **positional arguments**:*

```
>>> def f(x,y) :  
...     return x - y  
...  
>>> f(4,2)  
2  
>>> f(2,4)  
-2
```

*Functions can take a number of **arguments** as inputs, and these arguments can be provided in two ways:*

*2) as **keyword arguments** (this example with defaults):*

```
>>> def g(arg1=10, arg2=20) :  
...     return arg1 / float(arg2)  
...  
>>> g()  
0.5  
>>> g(1,20)  
0.05  
>>> g(arg2=100)  
0.1
```

*Python supports classes with **member attributes** and **functions**:*

```
>>> from math import pi
>>>
>>> class Circle() :
...     def __init__(self, r=1) :
...         self.radius = r
...     def area(self) :
...         return pi * (self.radius ** 2)
...
>>> c=Circle(4)
>>> c.radius
4
>>> c.area()
50.26548245743669
>>> 3.141592653589793 * 4 * 4
50.26548245743669
```

*As introduced on the last slide, the **import** statement avails library objects/functions:*

```
>>> import math
>>> math.pi
3.141592653589793
>>> from math import sin
>>> sin(math.pi/2)
1.0
>>> from math import *
>>> print e, log10(1000), cos(pi)
2.71828182846 3.0 -1.0
```

The three methods differ with respect to the interaction with the local namespace.

Python's syntax is (again) designed with clarity in mind, and good syntax is actually enforced by the interpreter.

*This comes from the fact that instead of curly braces or 'begin/end' keywords, code blocks are defined by **indentation**.*

This is unique to Python!

*A file with Python code in it is referred to as a **module**.*

Modules can be turned into executable scripts in two (or three) steps:

- 1) include the `if __name__ == '__main__':` block*
- 2) specify the interpreter (typically using a Unix shebang)*
- 3) *nix: make sure the file is executable (`chmod a+x <module.py>`)*

The screenshot on the next slide demonstrates both of these.

```
#!/usr/local/bin/python
from mrjob.job import MRJob

class MRHL(MRJob) :

    def mapper(self, _, line ) :
        lat,lon,src,nuid = line.rstrip().split(',')
        if src == 'physical' :
            yield nuid, (lon, lat)
        else :
            pass

    def reducer(self, nuid, lonlats ) :
        unique_lonlats = list(set([tuple(k) for k in lonlats]))
        yield nuid, len(unique_lonlats)

if __name__ == '__main__' :
    MRHL.run()
```

```
#!/usr/local/bin/python
import numpy as np

def run() :
    to_process = np.array([[1,2,3,4],[5,6,7,8]])
    for x in to_process.flatten() :
        print x*5

if __name__ == '__main__' :
    run()
```


Comments in Python are denoted by the '#' character.

```
# break when msg timestamp passes t_end
try:
    if created >= t_end:
        break

# if created DNE, keep going
except Exception as details:
    print details
    pass
```

*There are also special comments called **docstrings** that immediately follow class and function definitions.*

```
def function(parameters, options) :  
    """Do something really, really important """
```

Docstrings are denoted by triple quotes.

Useful for auto source documentation tools:

PyDoc, Dphinx, Doxygen

INTRO TO PYTHON

LAB