Introduction to Python

Installation, the basic idea, and some exploration

Daniel Guest

September 26, 2016

Installation

Anaconda - the easy way

1. Check to see if you already have Python — just type "Python" your terminal
2. Download Anaconda for your operating system (use the Python 3.5 version)
   - Windows: `https://repo.continuum.io/archive/Anaconda3-4.1.1-Windows-x86_64.exe` (64-bit) and `https://repo.continuum.io/archive/Anaconda3-4.1.1-Windows-x86.exe` (32-bit)
   - Mac OS: `https://repo.continuum.io/archive/Anaconda3-4.1.1-MacOSX-x86_64.pkg`
3. Install Anaconda
4. Anaconda is popular because it manages much of the difficult part of Python for you, and it includes most of the packages for scientific computing, data processing/visualization, and machine learning
5. Install any packages not included with Anaconda through the Anaconda GUI

If you install the hard way...

- Install from `https://www.python.org/downloads/release/python-352/`
- Learn how to install packages from `https://packaging.python.org/installing/`
- You'll probably want...
  1. NumPy — MATLAB style matrix and signal processing
  2. Matplotlib — plotting ala ggplot2
  3. pandas — data frames similar to R's data frames
  4. SciPy — all sorts of handy stuff for scientific computing
  5. PyMC3 — Bayesian models in Python
  6. Theano — Fast computation of complex mathematical expressions (used in deep learning)
  7. PyBrain — "easy" neural networks
- Check out the "SciPy Stack" for a good group of useful packages.

Why Python?

The design of Python

- Python is *readable* — the syntax is clean and designed for human eyes
- Python is *interpreted* — you can test small segments of code easily
- Python is *extensible* and *modular* — if basic Python doesn't do something you want, it's easy to install packages that do
- Python is *easy* — much of Python's scientific computing code was designed by scientists for scientists
- Python is *fast* — although not every built-in Python method is fast, there are many ways to speed up Python if you need it
- Python is *expressive* — your code will often be shorter and clearer
- Python is *general purpose* — you can do anything, all in one language

The design of Python

- Together, all of these characteristics mean that Python will make your life easier!
- Many people will tell you that Python is *natural* and *intuitive*, or that Python is the easiest way to express their thoughts in code
- As I recently saw someone on a forum write, "Python makes programming fun again"

The Python community

- One key advantage of Python is its community
- The community is *large*, *diverse*, and *active*, so most areas of your interest are probably covered, and it's easy to get help
- Much of the community follows the PEP 8 style guide (`https://www.python.org/dev/peps/pep-0008/`), leading to consistency in how code is written across the community
- The vast majority of Python code is *open source*, so you'll never have to pay for anything

Exploring Python Through the Terminal

Starting Python

- If you have Python anywhere on your computer, you can simply run "python" in your terminal
- But, no one really uses the plain Python terminal! For this demo, we'll use IPython, an enhanced Python terminal
- To run IPython, just run "ipython" in your terminal or command prompt
- The IPython terminal works very similarly to the terminal in R or MATLAB
- We can talk about writing scripts and using integrated development environments (IDEs) in the future — for now, we'll focus on getting the basics down!
- We'll move on once everyone has IPython running...

Variables

```
In [1]: x = 5

In [2]: x
Out[2]: ??

In [3]: x = x + 5

In [4]: x
Out[4]: ??

In [5]: x = "Hello"

In [6]: x
Out[6]: ??

In [7]: y = " World!"

In [8]: x + y
Out[8]: ??

In [9]: x = True

In [10]: y = False

In [11]: x and y
Out[11]: ??

In [12]: x or y
Out[12]: ??

In [13]: print(x)
Out[13]: ??
```

Variables

```
In [1]: x = 5

In [2]: x
Out[2]: 5

In [3]: x = x + 5

In [4]: x
Out[4]: 10

In [5]: x = "Hello"

In [6]: x
Out[6]: "Hello"

In [7]: y = " World!"

In [8]: x + y
Out[8]: "Hello world!"

In [9]: x = True

In [10]: y = False

In [11]: x and y
Out[11]: False

In [12]: x or y
Out[12]: True

In [13]: print(x)
Out[13]: True
```

Variables

```
In [14]: x = 5

In [15]: print(x)
Out[15]: ??

In [16]: y = "words"

In [17]: print(x + y)
Out[17]: ??

In [18]: print(str(x) + y)
Out[18]: ??

In [18]: print("I know at least " + str(x) + y)
Out[18]: ??
```

Variables

```
In [14]: x = 5

In [15]: print(x)
Out[15]: 5

In [16]: y = "words"

In [17]: print(x + y)
Out[17]: ERROR

In [18]: print(str(x) + y)
Out[18]: 5words

In [18]: print("I know at least " + str(x) + " " + y)
Out[18]: "I know at least 5 words"
```

Variables: notes

1. Values are assigned to a name by the = operator
2. Variables are *dynamic*, so they can store anything!
3. While powerful, dynamic variables can lead to errors if you don't keep track of what they contain!
4. String concatenation and boolean comparison is easy and readable
5. Convert non-strings to strings with str()
6. At least in terminal, just writing the name of a variable and passing it to the "print" function do the same thing

Strings

```
In [19]: x = "Hello, how are you?"

In [20]: x
Out[20]: ??

In [21]: x[0:5]
Out[21]: ??

In [22]: x[0:5] + " " + x[0:5] + "?"
Out[22]: ??

In [23]: len(x)
Out[23]: ??

In [24]: x.count("h")
Out[24]: ??

In [25]: x.find("h")
Out[25]: ??

In [26]: x[7]
Out[26]: ??

In [27]: x[x.find("a")]
Out[27]: ??

In [28]: x = "C:\Documents\lab\python\something_old.txt"

In [29]: x = x[0:-4] + ".py"

In [30]: x.replace("old", "new")
Out[30]: ??
```

Strings

```
In [19]: x = "Hello, how are you?"

In [20]: x
Out[20]: "Hello, how are you?"

In [21]: x[0:5]
Out[21]: "Hello"

In [22]: x[0:5] + " " + x[0:5] + "?"
Out[22]: "Hello Hello?"

In [23]: len(x)
Out[23]: 19

In [24]: x.count("h")
Out[24]: 1

In [25]: x.find("h")
Out[25]: 7

In [26]: x[7]
Out[26]: "h"

In [27]: x[x.find("a")]
Out[27]: "a"

In [28]: x = "C:\Documents\lab\python\something_old.txt"

In [29]: x = x[0:-4] + ".py"

In [30]: x.replace("old", "new")
Out[30]: "C:\Documents\lab\python\something_new.py"
```

Strings

1. Strings are very powerful and easy to manipulate in Python
2. You can concatenate strings using the + operator
3. You can index and slice strings in various ways — we'll see more of this later
4. There are many built-in functions to find and count characters and words inside of strings, or to replace parts of strings (we only scratched the surface here — string manipulation is much easier in Python than R or MATLAB)
5. You can check the length of a string with the len() — this is used for most objects in Python
6. Strings, and all other objects in Python, are indexed with [index] and not (index), which makes code more readable

Lists (or, where Python starts to get cool)

```
In [28]: x = [1, 2, 3, 4, 5]

In [29]: x
Out[29]: ??

In [30]: x[2]
Out[30]: ??

In [31]: x[0]
Out[31]: ??

In [32]: x[0:4]
Out[32]: ??

In [33]: x[-1]
Out[33]: ??

In [34]: x[-3:]
Out[34]: ??

In [35]: x[-4:-2]
Out[35]: ??

In [36]: x[5]
Out[36]: ??

In [37]: x.append(6)

In [38]: x[5]
Out[38]: ??
```

Lists (or, where Python starts to get cool)

```
In [28]: x = [1, 2, 3, 4, 5]

In [29]: x
Out[29]: [1, 2, 3, 4, 5]

In [30]: x[2]
Out[30]: 3

In [31]: x[0]
Out[31]: 1

In [32]: x[0:4]
Out[32]: [1, 2, 3, 4]

In [33]: x[-1]
Out[33]: 5

In [34]: x[-3:]
Out[34]: [3, 4, 5]

In [35]: x[-4:-2]
Out[35]: [2, 3]

In [36]: x[5]
Out[36]: ERROR

In [37]: x.append(6)

In [38]: x[5]
Out[38]: 6
```

Lists

```
In [43]: x = ["Joe"]

In [44]: len(x)
Out[44]: ??

In [45]: x = ["Austin", "Houston", "Dallas", "San Antonio", "Fort Worth"]

In [46]: len(x)
Out[46]: ??

In [47]: x.insert(1, "El Paso")

In [48]: x[0:3]
Out[48]: ??

In [49]: x.index["San Antonio"]
Out[49]: ??

In [50]: x.sort()

In [51]: x
Out[51]: ??

In [52]: x = ["9", 9, "Nine", ["9", 9, "Nine"], True]

In [53]: x[0] + x[2]
Out[53]: ??

In [54]: x[3]
Out[54]: ??

In [55]: x[1] + x[3][1]
Out[55]: ??
```

Lists

```
In [43]: x = ["Joe"]

In [44]: len(x)
Out[44]: 1

In [45]: x = ["Austin", "Houston", "Dallas", "San Antonio", "Fort Worth"]

In [46]: len(x)
Out[46]: 5

In [47]: x.insert(1, "El Paso")

In [48]: x[0:3]
Out[48]: ["Austin", "El Paso", "Houston"]

In [49]: x.index["San Antonio"]
Out[49]: 4

In [50]: x.sort()

In [51]: x
Out[51]: ["Austin", "Dallas", "El Paso", "Fort Worth", "Houston", "San Antonio"]

In [52]: x = ["9", 9, "Nine", ["9", 9, "Nine"], True]

In [53]: x[0] + x[2]
Out[53]: "9Nine"

In [54]: x[3]
Out[54]: ["9", 9, "Nine"]

In [55]: x[1] + x[3][1]
Out[55]: 18
```

Practical Python

Importing modules

Python, unlike MATLAB, has a small standard library. Although Python's standard library is very powerful, sometimes we need specific tools to get the job done. Python does this by allowing the user to import modules. We'll see how this is done below...

```
In [56]: x = [1, 4, 9, 16, 25]

In [57]: sqrt(x[1])
Out[57]: ??

In [58]: import math

In [59]: math
Out[59]: ??

In [60]: math.sqrt(x[1])
Out[60]: ??

In [61]: dir(math)
Out[61]: ??

In [62]: import math as m

In [63]: m.sqrt(x[4])
Out[63]: ??

In [64]: from math import sqrt

In [65]: sqrt(x[4])
Out[65]: ??
```

Modules — notes

1. Modules are bundles of related code
2. You have to explicitly import modules to use them by using the *import* keyword
3. This might seem tedious at first, but it (a) allows for functions to share names, if they come from different modules, (b) makes it easy to import your own functions across different projects, and (c) makes larger projects more organized than in other languages

A relatable example — the idea

Suppose we want to...

1. Import a .wav file of a recorded vowel
2. Calculate the spectrum of the vowel
3. Plot the spectrum of the vowel and add labels

A relatable example — the code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

location = "/home/daniel/Documents/lab/vowgenface/vowels/unswapped/kabrii01.wav"

[fs, vowel] = wavfile.read(location)
n = len(vowel)
time_axis = np.arange(n)/fs
vowel = vowel/np.max(np.abs(vowel))

plt.plot(time_axis, vowel)
plt.ylabel("Amplitude")
plt.xlabel("Time")
plt.show()

log_spectrum = 20*np.log10(np.fft.rfft(vowel))
dt = n/fs
freq_axis = (np.arange(n)/dt)[0:len(log_spectrum)]

plt.plot(freq_axis, log_spectrum)
plt.ylabel("Amplitude")
plt.xlabel("Frequency")
plt.show()
```

A relatable example — notes

1. Usually, modules are imported at the top of the code
2. NumPy and PyPlot are designed to be familiar to MATLAB users (easy transition)

An advanced example — the idea

1. Monte Carlo Markov chain (MCMC) algorithms work by using a random walk to sample a posterior distrobution that cannot be solved analytically
2. Suppose we want to see this random walk in action to develop a better understanding of what's really going on!
3. We can simulate some data using NumPy or Pandas...
4. Then, we can use PyMC3 to create a Bayesian model...
5. Finally, we can use matplotlib to make an animation of each step in action and save it to a video file for easy sharing!
6. We're not going to write this example — we're just going to look at the code for an idea of what Python can do very easily

Thoughts and resources

Some thoughts...

1. I love Python, but even I don't use Python for everything — I still prefer R+dplyr for data manipulation and R+ggplot2 for data visualization
2. That said, Python is catching up in every area that it's behind, and is ahead in many areas
3. Python is much better than MATLAB/R for machine learning and advanced Bayesian modeling, for example
4. Python enables you to focus on just one language, rather than being split between multiple languages
5. Python is useful at every skill level and scale — it's easy for a beginner, while still being powerful for an expert

Some resources...

1. Type "help(function)" into the Python terminal to get built-in help!
2. `https://www.python.org/about/gettingstarted/` — great documentation!
3. `https://wiki.python.org/moin/BeginnersGuide/Programmers` — lots of tutorials, videos, guides, example code, etc.
4. `https://docs.python.org/3/` — formal documentation, for the technically adventurous