

# Python Introduction

Rosie Martinez

(credit E. Desmond, F. Burkholder, K. Thomas)

# Learning Objectives

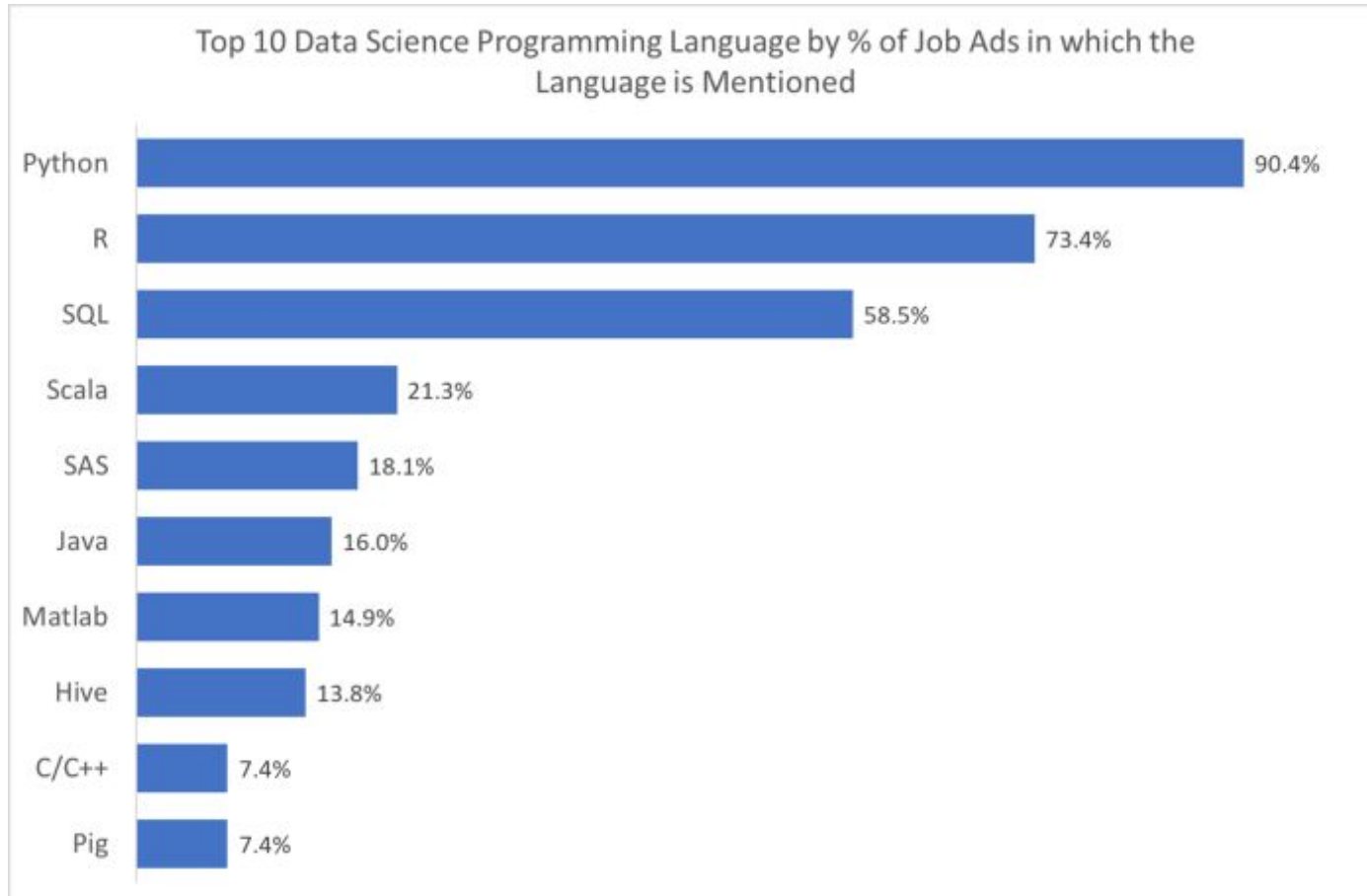
- By the end of this lecture you will:
  - Have a better understanding of the work environment for coding in Python
  - Understand the coding workflow
  - Review the data types and data structures
  - Start implementing Python best practices

# Python... for Data Science?

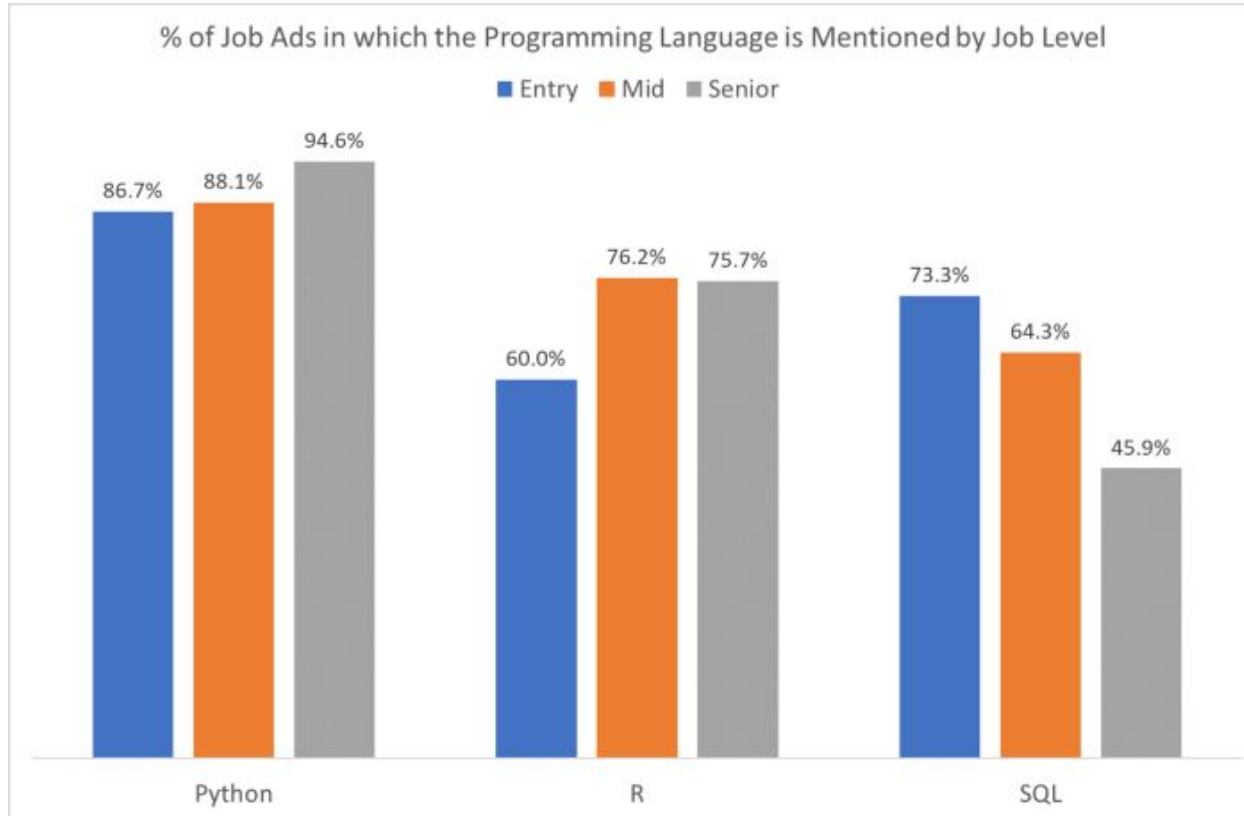
Why do we use Python?

- General purpose programming language that supports many paradigms
- Efficient Syntax
  - Fewer lines of code needed to do the same thing relative to C++ or Java
  - Designed to emphasize readability (E.g. uses white space to define block rather than brackets)
  - [Zen of Python](#)
- Large Development Community
  - Open Source Development ([Python on Github](#))
- Growing Popularity
  - Big Data
  - Data Science

# Python for Data Science!



# Python for Data Science!



# Python 2 vs. 3

Principle of Python 3: "reduce feature duplication by removing old ways of doing things"

Python 2	Python 3
<code>print 'Hello World'</code>	<code>print('Hello World')</code>
<code>3 / 2 = 1, (3 / 2.) = 1.5</code>	<code>3 / 2 = 1.5, 3 // 2 = 1</code>
types: <code>str()</code> , <code>unicode()</code>	type: <code>str()</code>
<code>range(n)</code> - makes a list <code>xrange(n)</code> - iterator	<code>range(n)</code> - iterator <code>list(range(n))</code> - makes a list
<code>.items()</code> - makes a list <code>.iteritems()</code> - iterator	<code>.items()</code> - iterator
<code>map()</code> - makes a list	<code>map()</code> - map object <code>list(map())</code> - makes a list
<code>my_generator.next()</code>	<code>next(my_generator)</code>

# Python 2 Deprecated

## Sunsetting Python 2

We are volunteers who make and take care of the Python programming language. We have decided that January 1, 2020, was the day that we sunset Python 2. That means that we will not improve it anymore after that day, even if someone finds a security problem in it. You should upgrade to Python 3 as soon as you can.

- If you need to work in both (which you shouldn't)
  - use [conda](#) to create an environment in your Anaconda distribution
    - `$ conda create -n py2 python=2 anaconda` (if you have Python 3 installed)
    - `$ conda create -n py3 python=3 anaconda` (if you have Python 2 installed)
    - `$ source activate py2` (or `py3`)
    - `$ source deactivate`
- Galvanize curriculum used to be all Python 2

<https://www.python.org/doc/sunset-python-2/>

# Python work environment

Options differ in complexity and abstraction from what is absolutely required to write code.



# Python work environment

Options differ in complexity and abstraction from what is absolutely required to write code.

**Simplest:** Use a Terminal text editor like Vim or Nano to write script (.py files), then execute the script from Terminal, e.g. `python script.py datafile.csv`

Advantages:

- Ensures that code works sequentially and as a cohesive whole.
  - Doesn't maintain a Namespace (like IPython or Jupyter Notebook)
- Will always work! (And this is the environment you have when you are on a server)
- [Vim](#) and [tmux](#) can get you text editor, IPython console, and Terminal all in one screen.

Disadvantages:

- Debugging more difficult. Need to learn a debugger (`pdb`) or use print statements to understand the state of the program.
- Visually not user-friendly for large projects.
- Vim difficult to learn.

# Python work environment

Options differ in complexity and abstraction from what is absolutely required to write code.

**Most Complex:** Use an Integrated Development Environment like Spyder.

The screenshot displays the Spyder Python IDE interface. The main window is divided into several panes:

- Project Explorer:** Shows the file structure of the project, including folders like 'data', 'img\_src', 'requirements', 'scripts', 'tests', and 'workers'.
- Code Editor:** Contains Python code for data analysis and plotting. The code includes imports for numpy, scipy, and matplotlib, and performs calculations and plotting.
- Variable Explorer:** Displays the variables defined in the code, including their names, types, sizes, and values. For example, 'array\_int8' is of type 'int8' with size (2, 1) and value [7].
- Python Console:** Shows the execution output of the code, including the creation of a 3D surface plot and a 2D polar plot.

The code in the editor is as follows:

```
6
7 import pylab
8 from numpy import cos, linspace, pi, sin, random
9 from scipy.interpolate import splprep, splev
10
11 ## Generate data for analysis
12
13 ## Make ascending spiral in 3-space
14 t = linspace(0, 1.75 * 2 * pi, 100)
15
16 x = sin(t)
17 y = cos(t)
18 z = t
19
20 ## Add noise
21 x = random.normal(scale=0.1, size=x.shape)
22 y = random.normal(scale=0.1, size=y.shape)
23 z = random.normal(scale=0.1, size=z.shape)
24
25
26 ## Perform calculations
27
28 ## Spline parameters
29 smoothness = 3.0 # Smoothness parameter
30 k_param = 2 # Spline order
31 nests = -1 # Estimate of number of knots needed (-1 = maximal)
32
33 ## Find the knot points
34 knot_points, u = splprep([x, y, z], s=smoothness, k=k_param, nests=-1)
35
36 ## Evaluate spline, including interpolated points
37 xnew, ynew, znew = splev(linspace(0, 1, 400), knot_points)
38
39
40 ## Plot results
41
42 ## TODO: Rewrite to avoid code smell
43 pylab.subplot(2, 2, 1)
44 data = pylab.plot(x, y, 'bo-', label='Data with X-Y Cross Section')
45 fit = pylab.plot(xnew, ynew, 'r-', label='Fit with X-Y Cross Section')
46 pylab.legend()
47 pylab.xlabel('x')
48 pylab.ylabel('y')
49
50 pylab.subplot(2, 2, 2)
51 data = pylab.plot(x, z, 'bo-', label='Data with X-Z Cross Section')
52 fit = pylab.plot(xnew, znew, 'r-', label='Fit with X-Z Cross Section')
53 pylab.legend()
54 pylab.xlabel('x')
```

The variable explorer shows the following variables:

Name	Type	Size	Value
array_int8	int8	(2, 1)	Min: 7 Max: 6
array_uint32	uint32	(2, 2, 3)	Min: 1 Max: 7
bars	container.BarContainer	20	BarContainer object of matplotlib.conta...
df	DataFrame	(3, 2)	Column names: bools, ints
filename	str	1	C:\ProgramData\Anaconda3\lib\site-packa...
list_test	list	2	[DataFrame, Numpy array]
nrows	int	1	344
r	float64	1	7.61180258934796
radii	float64	(20,)	Min: 0.4983836638535687 Max: 9.05648076942551
region	tuple	2	(slice, slice)
rgb	float64	(45, 45, 4)	Min: 0.0 Max: 1.0
series	Series	(1,)	Series object of pandas.core.series mod...
test_none	NoneType	1	NoneType object of builtins module

The Python console shows the following output:

```
...: ls = LightSource(270, 45)
...: # To use a custom hillshading mode, override the built-in shading
...: # in the rgb colors of the shaded surface calculated from 'shade':
...: rgb = ls.shade(z, cmap=cm.gist_earth, vert_exag=0.1, blend_mode='soft')
...: surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=rgb,
...:                        linewidth=0, antialiased=False, shade=False)
...:
...: plt.show()
```

The console also displays the execution of the code, including the creation of a 3D surface plot and a 2D polar plot.

# Python work environment

Options differ in complexity and abstraction from what is absolutely required to write code.

**Most Complex:** Use an Integrated Development Environment like Spyder.

Advantages:

- You can see a lot of what's going on (as long as you know where to look).
  - Variable values
- You (mostly) get consistent behavior from the application independent of Operating System.

Disadvantages:

- Learning the IDE takes time (and right now isn't your time better spent learning Python?)
- Student gets addicted, and can't work from Terminal. (Developer at Pivotal: **IDEs are earned**)
- They don't work perfectly (is the problem with the Namespace, the IDE, your code?)

# Python work environment (strongly recommended)

Microsoft's Visual Studio Code

It's a "lightweight" text editor with a few IDE features:

- Integrated Terminal (that seems to work like an actual Terminal)
- Python linting
- Debugger
- Git integration
- Open source
- Visual sugar for software developers
- In fact, it's the [most popular development environment](#) with software developers.

# Jupyter Notebooks are **NOT** a work environment

What's good about them:

- Nice visual interface.
- Good REPL (Read-Evaluate-Print-Loop)
- Can mix code, plots, mathematical equations, clickable “right” answers.
  - Great for teaching
- A lot of examples out there (Kaggle submittals)

What's so *bad* about them:

- Don't have to program sequentially - leads to disorganized thinking.
- Can't use them to deploy anything.
- Maintained Namespace makes it easier to code, but harder to write good code that works as a cohesive whole.
- Horrible at version control in git
- Realistically, the audience for your Notebook is small
  - Developers: will want scripts
  - Managers and clients: will want reports

# Suggested workflow

- In a script, start with an `if __name__ == '__main__':` block (INEMB).
  - for now, just trust me - I'll demonstrate why in a bit
- `import` whatever you need to above the INEMB, start writing code below.
- In the iPython console, `run` your code and then check to see if you are getting values you expect.
- If you are getting values you expect, start encapsulating your code into functions (and later classes) above the INEMB.
- `import` these functions (and/or classes) into lpython to make sure they work.
- Demo - make a function that makes a deck of cards
  - Show usefulness of INEMB when importing `deck.py` into a `game.py` file.

# Python data types and data structures

Data types

Data structures

TYPE	DESCRIPTION	EXAMPLE VALUE(S)
int	integers	1, 2, -3 ...
float	real numbers, floating values	1.0, 2.5, 102342.32423 ...
str	strings	'abc'
tuple	an immutable tuple of values, each has its own type	(1, 'a', 5.0)
list	a list defined as an indexed sequence of elements	[1, 3, 5, 7]
dict	a dictionary that maps keys to values	{ 'a' : 1, 'b' : 2 }
set	a set of distinct values	{1, 2, 3}

# More on data structures

- **Lists:** ordered, dynamic collections that are meant for storing collections of data about disparate objects (e.g. different types). Many list methods. (type list)
- **Tuples:** ordered, static collections that are meant for storing unchanging pieces of data. Just a few methods. (type tuple)
- **Dictionaries:** unordered collections of key-value pairs, where each key has to be unique and immutable (type dict) Hash map associates key with the memory location of the value so lookup is fast.
- **Sets:** unordered collections of unique keys, where each key is immutable (type set). Hash map associates key with membership in the set, so checking membership in a set is fast (much faster than a list).



# Demo and breakouts

`data_structures.ipynb`

# Python good practice

- **PEP8:** [Style guide for Python](#). Addresses spacing, variable names, function names, line lengths. Highlights:
  - variable and function names are snake\_case, classes CamelCase
  - avoid extraneous whitespace
  - lines not longer than 79 characters
  - documentation!
  - can check if your code conforms to pep8:
    - `$ pycodestyle script.py`
- **Pythonic code:** [A guideline](#)
  - use `for` loops instead of indexing into arrays
  - use `enumerate` if you need the index
  - use `with` statements when working with files
  - use list comprehensions
  - `(if x:)` instead of `(if x == True:)`
  - and many others (see guide, and we'll address later in course)

# Breakout - pair up

Look through [PEP8](#) and [HitchHiker's Guide to Python Code Style](#) and make a list of 5 style suggestions that you weren't aware of. You have 5 minutes, then you'll be asked to share them with the class.

# Zen of Python

From within ipython:

```
In [1]: import this
```

The Zen of Python, by Tim Peters

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.

...

# Zen of Python examples

## Explicit is better than implicit

```
def make_dict(*args):  
    x, y = args  
    return dict(**locals())
```

```
def make_dict(x, y):  
    return {'x': x, 'y': y}
```

Which one is more explicit?

# Zen of Python examples

## Explicit is better than implicit

```
def make_dict(*args):  
    x, y = args  
    return dict(**locals())
```

```
def make_dict(x, y):  
    return {'x': x, 'y': y}
```

Which one is more explicit?



# Zen of Python examples

## **Sparse is better than dense**

```
if x == 1: print('one')
```

```
if x == 1:
```

```
    print('one')
```

```
if (<complex comparison 1> and  
    <complex comparison 2>):  
    # do something
```

```
cond_1 = <complex comparison 1>  
cond_2 = <complex comparison 2>  
if (cond_1 and cond_2):  
    # do something
```

Which is more sparse?

# Zen of Python examples

## Sparse is better than dense

```
if x == 1: print('one')
```

```
if x == 1:
```

```
    print('one')
```

```
if (<complex comparison 1> and  
    <complex comparison 2>):  
    # do something
```

```
cond_1 = <complex comparison 1>  
cond_2 = <complex comparison 2>  
if (cond_1 and cond_2):  
    # do something
```

Which is more sparse?





# Line continuation

```
french_insult = \  
"Your mother was a hamster, and \  
your father smelt of elderberries!"
```

```
french_insult = (  
"Your mother was a hamster, and "  
"your father smelt of elderberries!"  
)
```

Preferred?

# Line continuation

```
french_insult = \  
"Your mother was a hamster, and \  
your father smelt of elderberries! "
```

```
french_insult = (  
"Your mother was a hamster, and "  
"your father smelt of elderberries!"  
)
```



Preferred?

# Tips for speeding up your Python code

First, do you really need to do it?

**“Premature optimization is the root of all evil” - [Donald Knuth](#)**

Get your code working first, then go back and refactor it to improve it.

# Tips for speeding up your Python code

First, do you really need to do it?

**“Premature optimization is the root of all evil”** - [Donald Knuth](#)

Get your code working first, then go back and refactor it to improve it.

Ok, you still want to do it....

From “5 Tips to speed up your Python code” by Bob at PyBites:

<https://pybit.es/faster-python.html>

# Speed up tip #1

## **Know the data structures**

“...it is often a good idea to use sets or dictionaries instead of lists in cases where:

- The collection will contain a large number of items
- You will be repeatedly searching for items in the collection
- You do not have duplicate items”

- Hitchhiker's Guide to Python

# Speed up tip #2

## Reduce memory footprint

```
msg = "The is an\n"  
msg += "inefficient way\n"  
msg += "(memory-wise)\n"  
msg += "to make a multiline string.\n"
```

## Better:

```
msg = ["This is a", "much better", "way to", "do it."]  
'\n'.join(msg)
```

# Speed up tip #3

## Use built-in functions and libraries

- Builtin functions like `sum`, `max`, `any`, `map`, etc. are implemented in C. They are very efficient and well tested.
- The `collections` module has the `defaultdict` and `Counter` builtins

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
```

```
>>> Counter('mississippi')
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

# Speed up tip #4

## Move calculations outside the loop

(And don't do a calculation or operation more than once if you don't have to.)

### Inefficient:

```
accuracies = []  
for i in range(10):  
    clf = RandomForestClassifier()  
    clf.fit(X_train[i], y_train[i])  
    y_pred = clf.predict(X_test)  
    accuracies.append(accuracy_score(y_true, y_pred))
```



# Speed up tip #4

## Move calculations outside the loop

(And don't do a calculation or operation more than once if you don't have to.)

### Better:

```
clf = RandomForestClassifier()
accuracies = []
for i in range(10):
    clf.fit(X_train[i], y_train[i])
    y_pred = clf.predict(X_test)
    accuracies.append(accuracy_score(y_true, y_pred))
```

# Speed up tip #5

## Keep your code base small

Think about how much code you're importing when you import from your personal modules. Are you pulling in many classes or functions just to get a few that you need?

Several separate scripts containing Class definitions are better than one massive script containing all the classes. It takes time to read in all the functions and classes!

Use an **INEMB**.

# Learning Objectives

- By the end of this lecture you will:
  - Have a better understanding of the work environment for coding in Python
  - Understand the coding workflow
  - Review the data types and data structures
  - Start implementing Python best practices