Python Introduction

Objectives

- Python - brief history
- Python popularity, and use in Data Science
- Python 2 vs. 3
- Work environment
- Workflow
- Data types and data structures
- Python good practice
- Zen of Python examples
- Tips for speeding up your Python code

Python - a brief history

- In the late 1980's Guido van Rossum conceived and started to implement Python as a successor to the ABC programming language. Guido said he needed "a decent scripting language." Python itself named from *Monty Python's Flying Circus.*

- In 1994 Python 1.0 was released. Some functional programming tools - lambda, reduce(), filter(), map() - were added courtesy of a Lisp hacker.

- Python 2 was released in 2000 with the help of a more transparent, community-based development process (Python Software Foundation). List comprehensions and generators were introduced.

- Python 3 was released in 2008. Had an emphasis on removing duplicative constructs and modules. It's a backward incompatible release, though many of its major features have been back-ported to Python 2.

- EOL date for Python 2 was originally set for 2015, been extended to 2020. Numpy will not be supporting Python 2 in 2020.

Guido, named BDFL by Python community, but he has abdicated his throne!

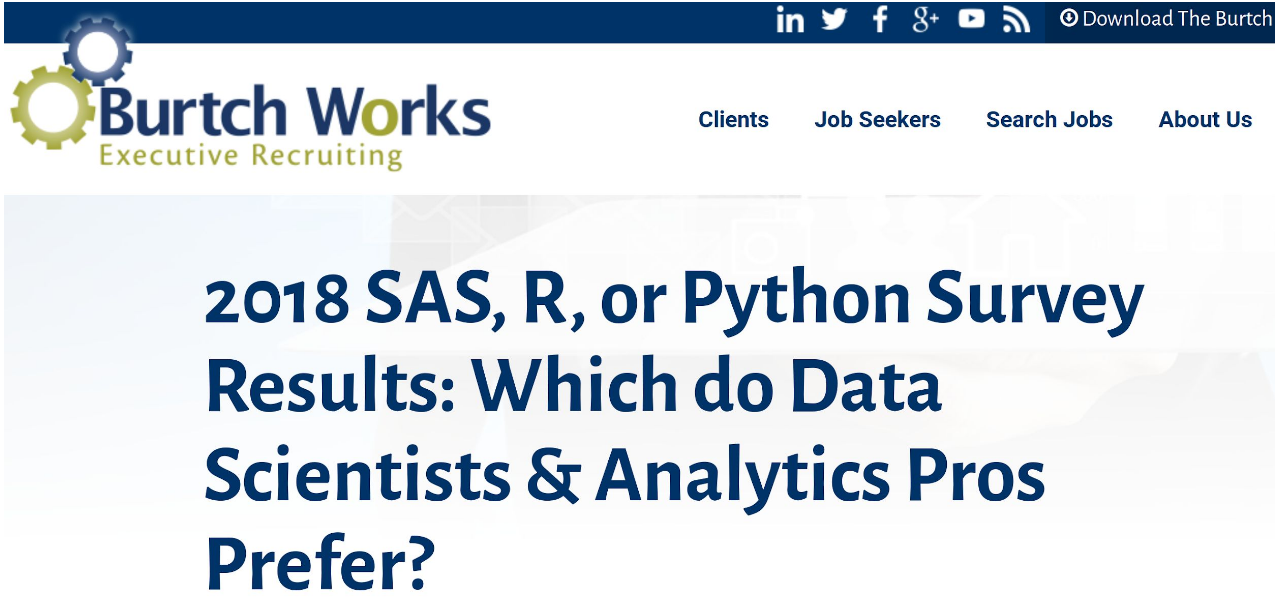
https://en.wikipedia.org/wiki/History_of_Python

Python popularity

- General purpose programming language that supports many paradigms
    - imperative, object-oriented, functional
- Interpreted, instead of compiled
    - has rapid REPL (Read-Evaluate-Print-Loop)
- Design philosophy emphasizes code readability
    - white space rather than brackets/braces determine code blocks
    - Zen of Python
- Efficient syntax
    - fewer lines of code needed to do the same thing relative to C++, Java
- Large development community
    - large and comprehensive standard library (NumPy, SciPy, MatplotLib, Pandas, Scikit-Learn)
    - open-source development (Python on Github)

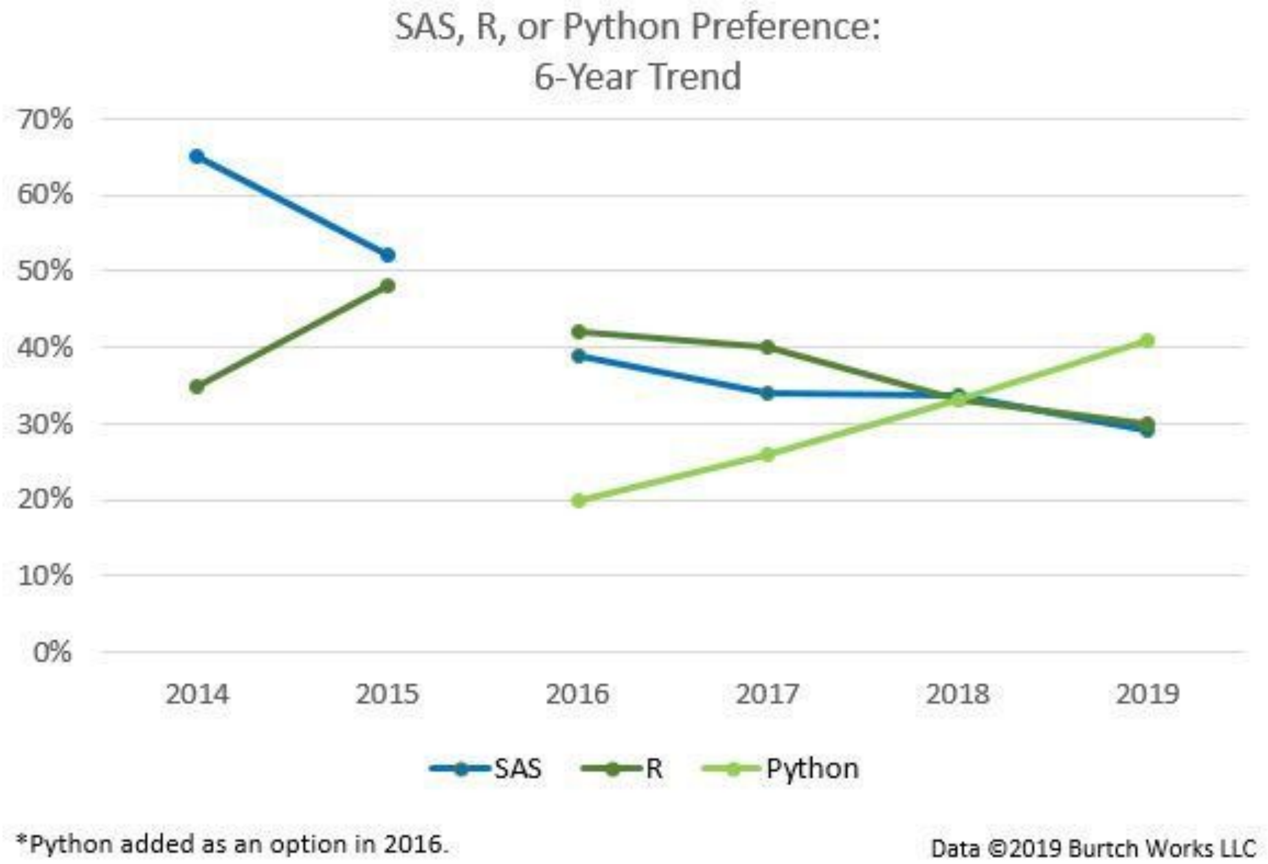
https://en.wikipedia.org/wiki/Python_(programming_language)

Python for Data Science



[What is SAS?] [What is R?] [source]

Python for Data Science



source

Python 2 vs. 3

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

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

http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

Python 2 vs. 3

- You may need to work in both
  - 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

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.

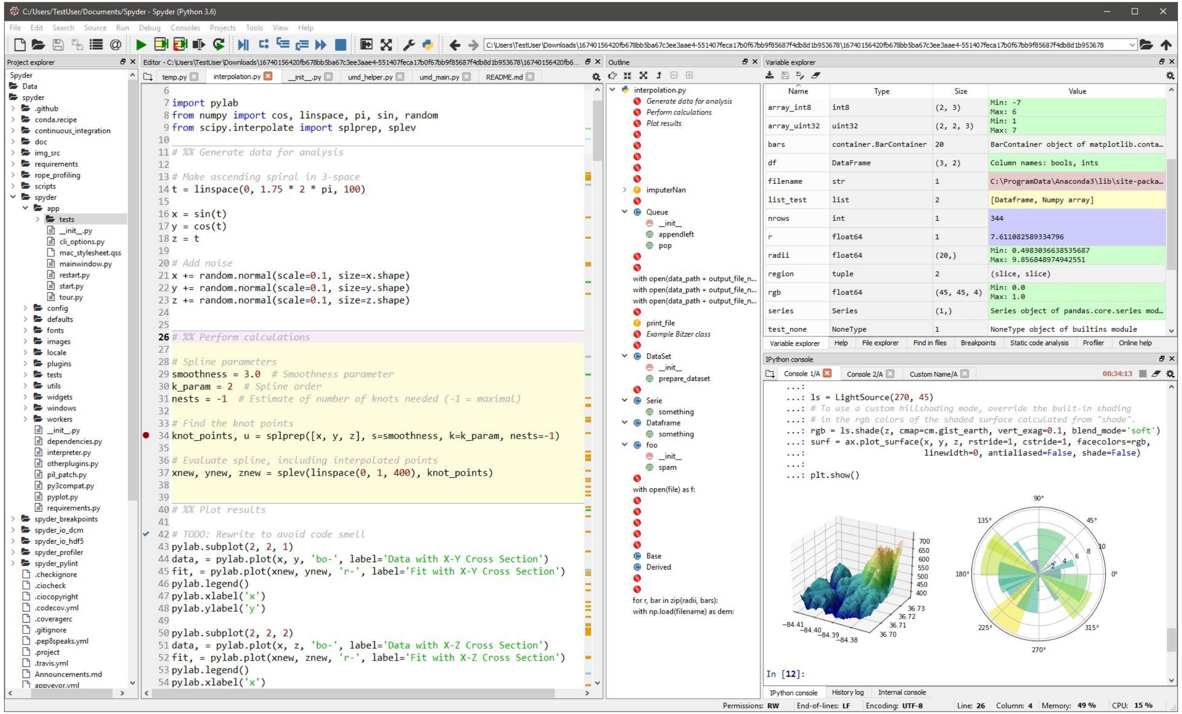
Disadvantages:

- 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.

Complex: Use an Integrated Development Environment like Spyder.



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).
- `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 Ipython to make sure they work.

INEMB example - deck.py

```python
import random

def shuffle(deck):
    random.shuffle(deck)

def shuffle_and_draw(deck):

    shuffle(deck)

    return deck.pop()

def make_deck(suits, numbers, faces):
    vals = numbers + faces
    deck = [str(val) + ' of ' + suit for val in vals for suit in suits]
    return deck

if __name__=='__main__':
    suits = ['hearts', 'clubs', 'diamonds', 'spades']
    numbers = list(range(2,11))
    faces = ['J', 'Q', 'K', 'A']
    vals = numbers + faces
    deck = [str(val) + ' of ' + suit for val in vals for suit in suits]
    print("Here are the first 10 cards in the deck I made after the if-name-main block!")
    print(deck[:10])
    deck2 = make_deck(suits, numbers, faces)
    if deck2 == deck:
        print("The decks are the same!")
    else:
        print("Wrong, try again.")
```

INEMB example - game.py

```
from deck import make_deck, shuffle, shuffle_and_draw
import random

def play_war(deck):

    card_1 = shuffle_and_draw(deck)
    card_2 = shuffle_and_draw(deck)

    print(f'''
    Card 1: {card_1}
    Card 2: {card_2}
    ''')

if __name__ == '__main__':

    suits = ['hearts', 'clubs', 'diamonds', 'spades']
    numbers = list(range(2,11))
    faces = ['J', 'Q', 'K', 'A']

    deck = make_deck(suits, numbers, faces)

    play_war(deck)
```

Python data types and data structures

	TYPE	DESCRIPTION	EXAMPLE VALUE(S)
Data types	`int`	integers	`1, 2, -3` ...
	`float`	real numbers, floating values	`1.0, 2.5, 102342.32423` ...
	`str`	strings	`'abc'`
Data structures	`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**.

Objectives

- Python - brief history
- Python popularity, and use in Data Science
- Python 2 vs. 3
- Work environment
- Workflow
- Data types and data structures
- Python good practice
- Zen of Python examples
- Tips for speeding up your Python code