Python: What's in it For You

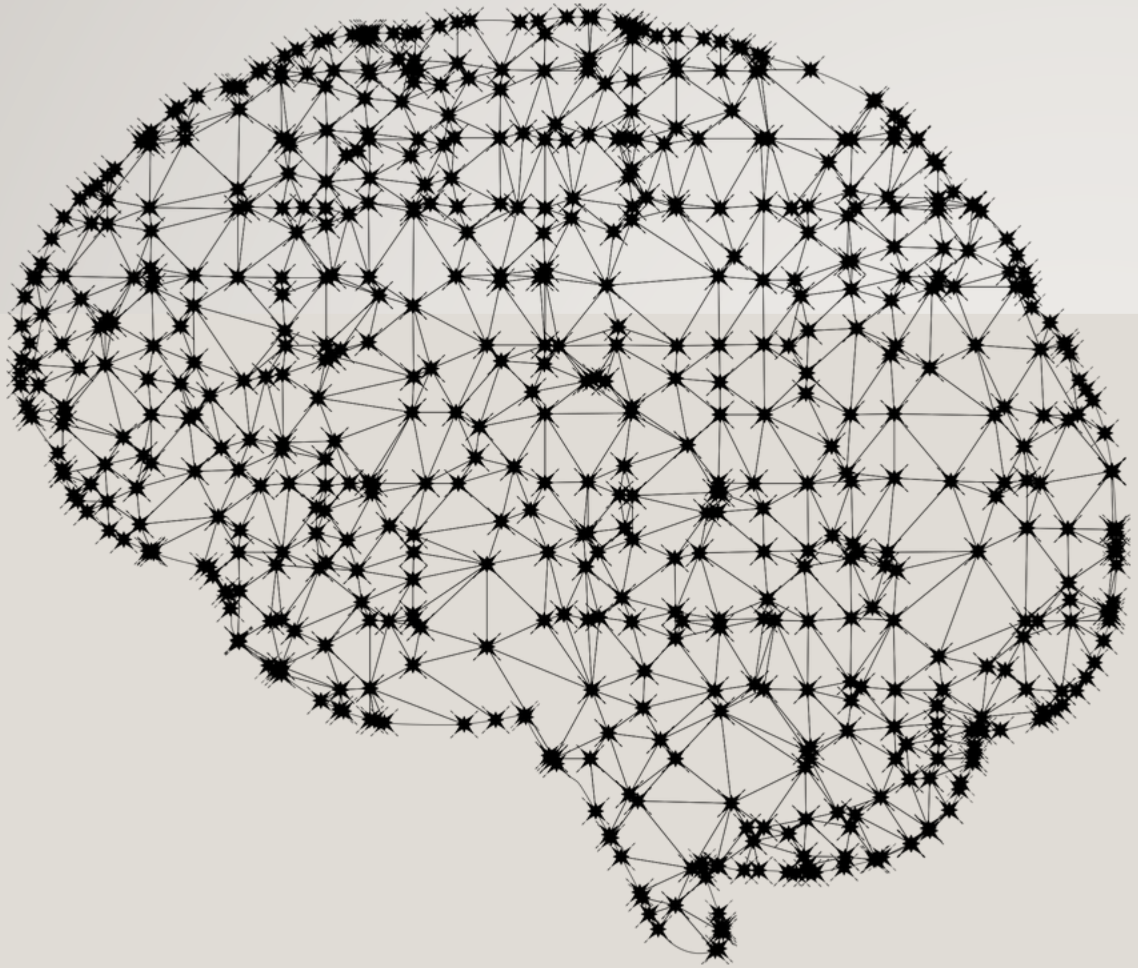
Douglas Starnes

CodeStock 2019

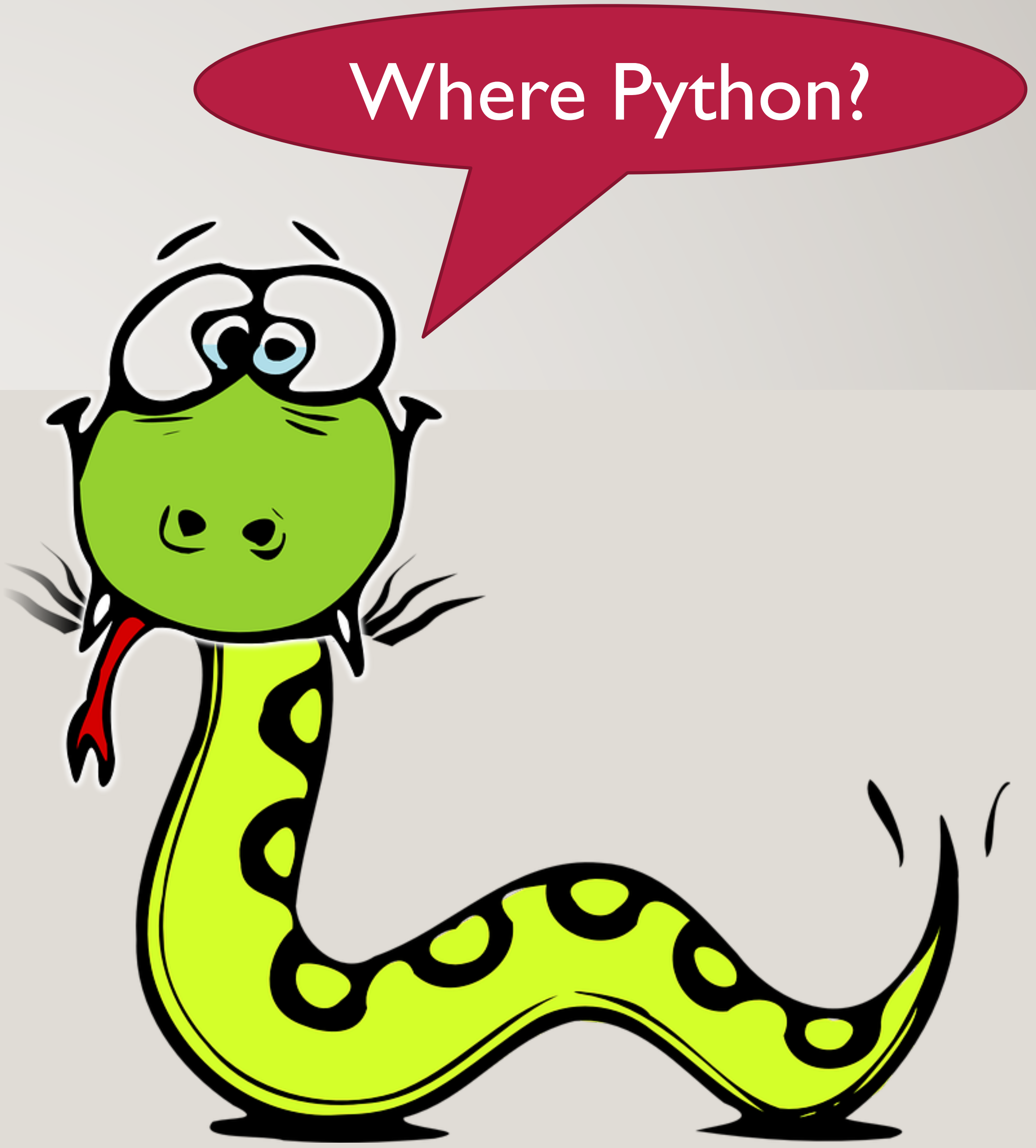
Why Python?

- Can ‘keep it in your head’
- It’s ‘close to the metal’
- Uses in a wide variety of fields
- Open source, cross platform and free
- Become dangerous in a weekend, and useful in a week

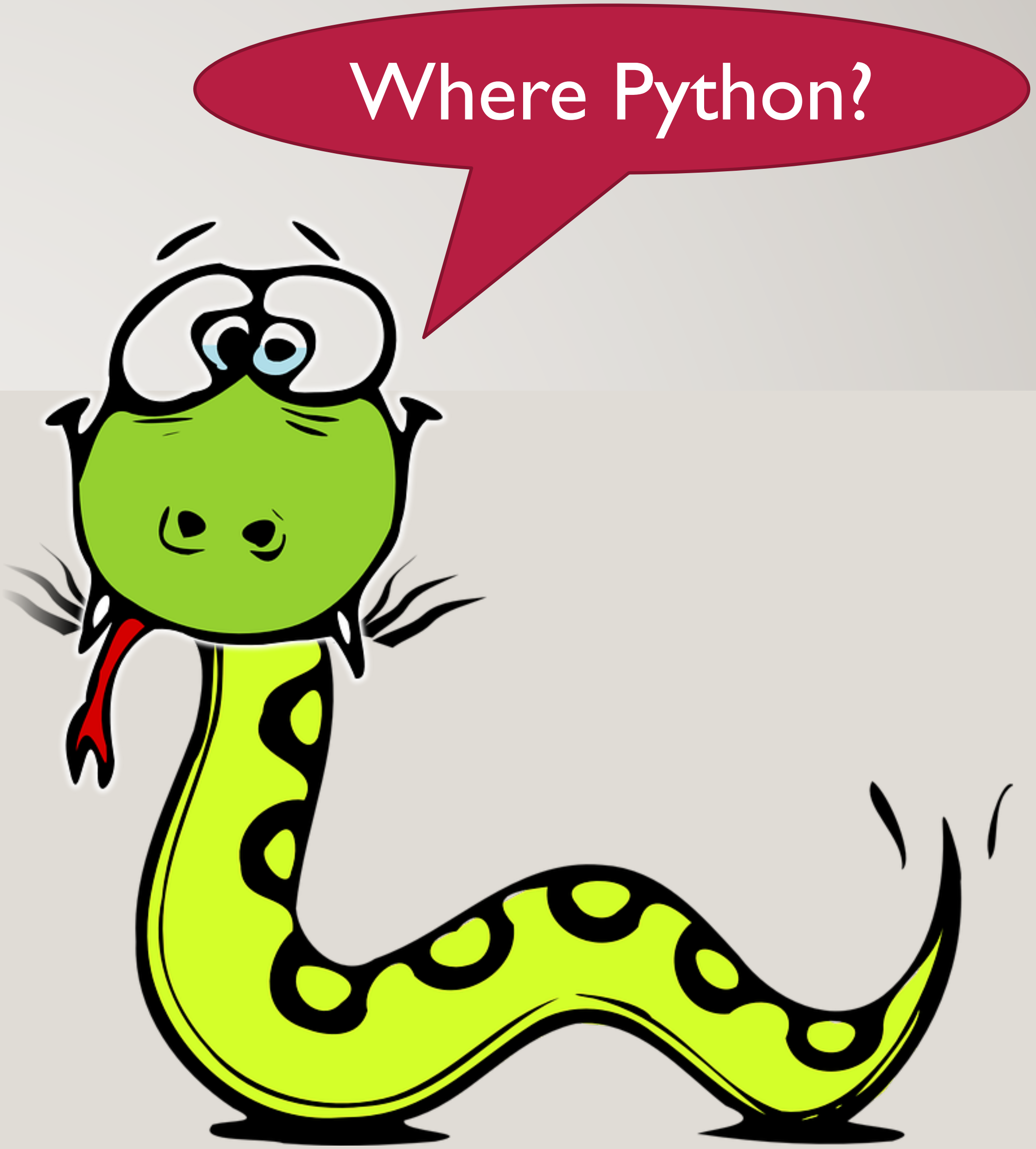
@poweredbyaltnet

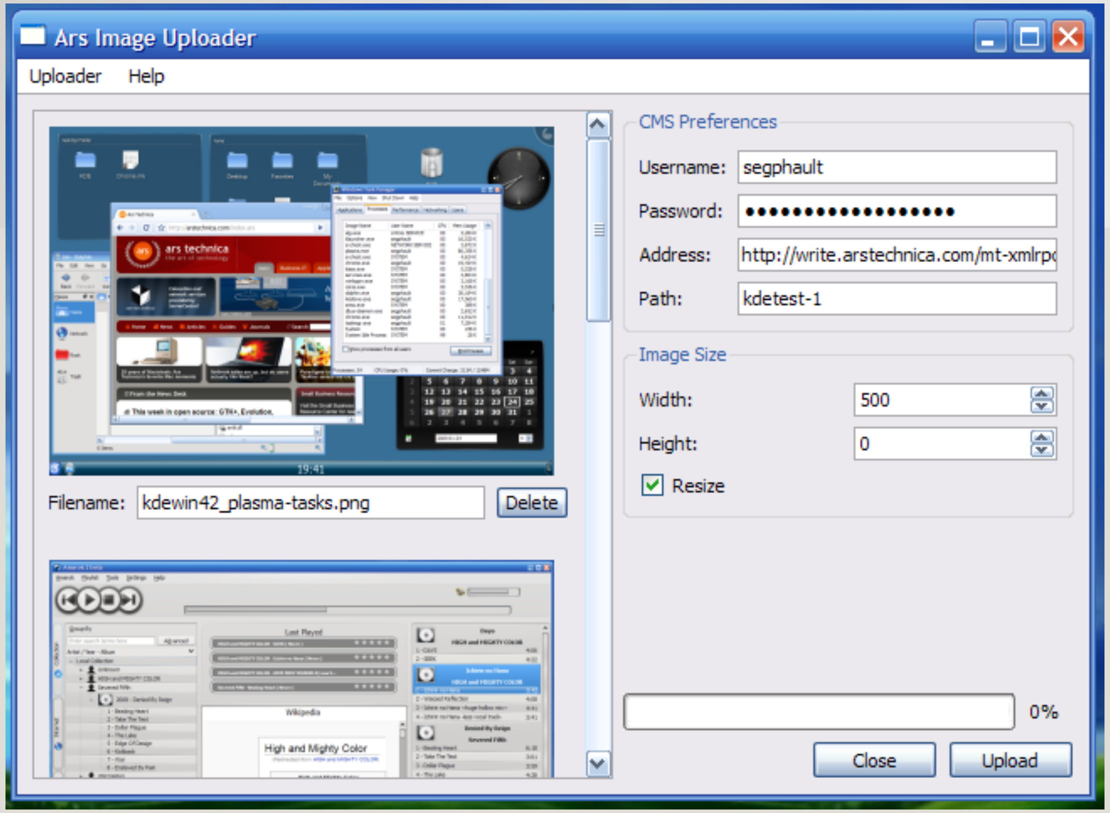


Data Science & Machine Learning



Web Applications





CMS Preferences

Username: segphault
Password: ••••••••••••••••••
Address: http://write.arstechnica.com/mt-xmlrpc
Path: kdetest-1

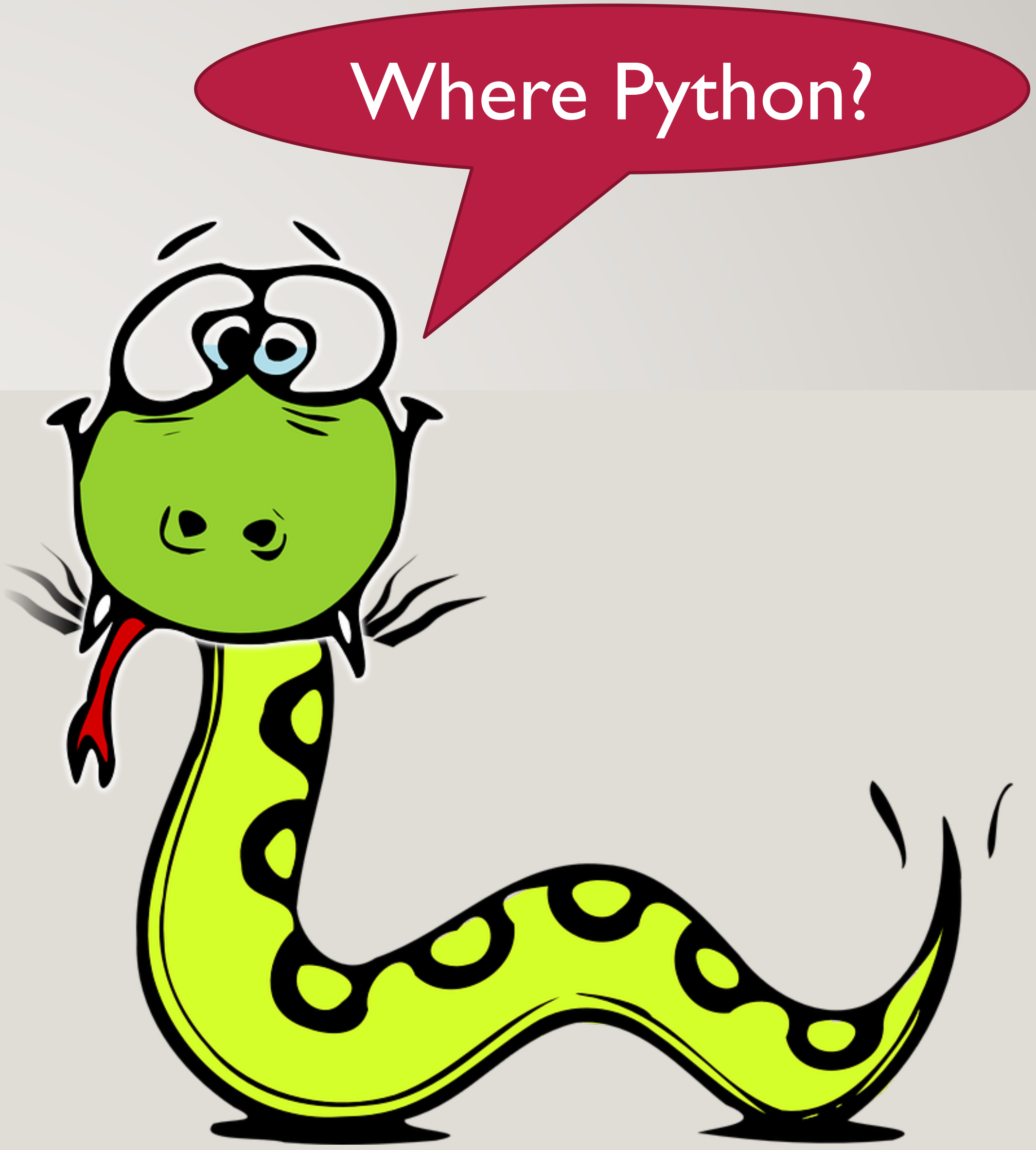
Image Size

Width: 500
Height: 0
✓ Resize

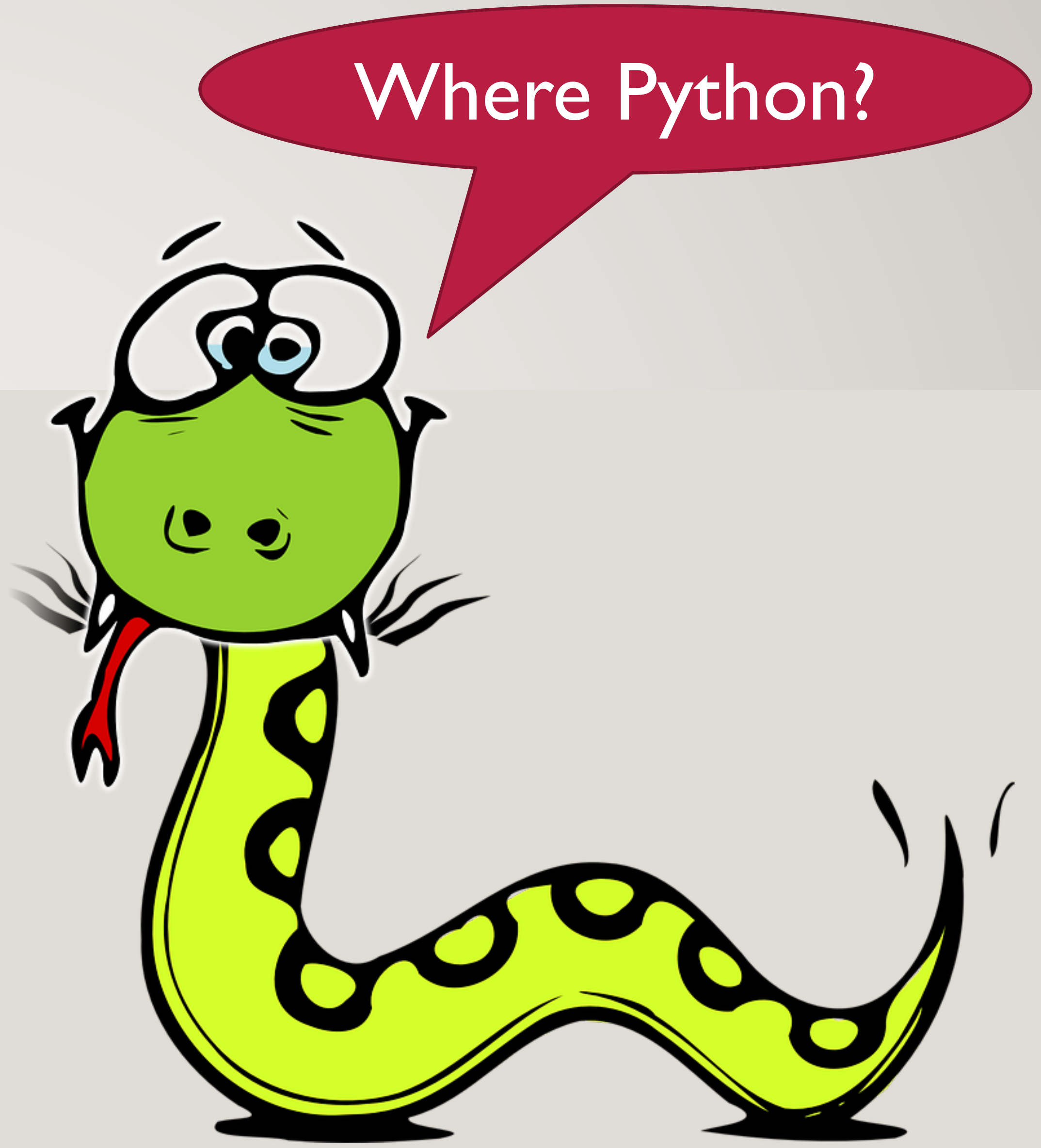
Filename: kdewin42_plasma-tasks.png

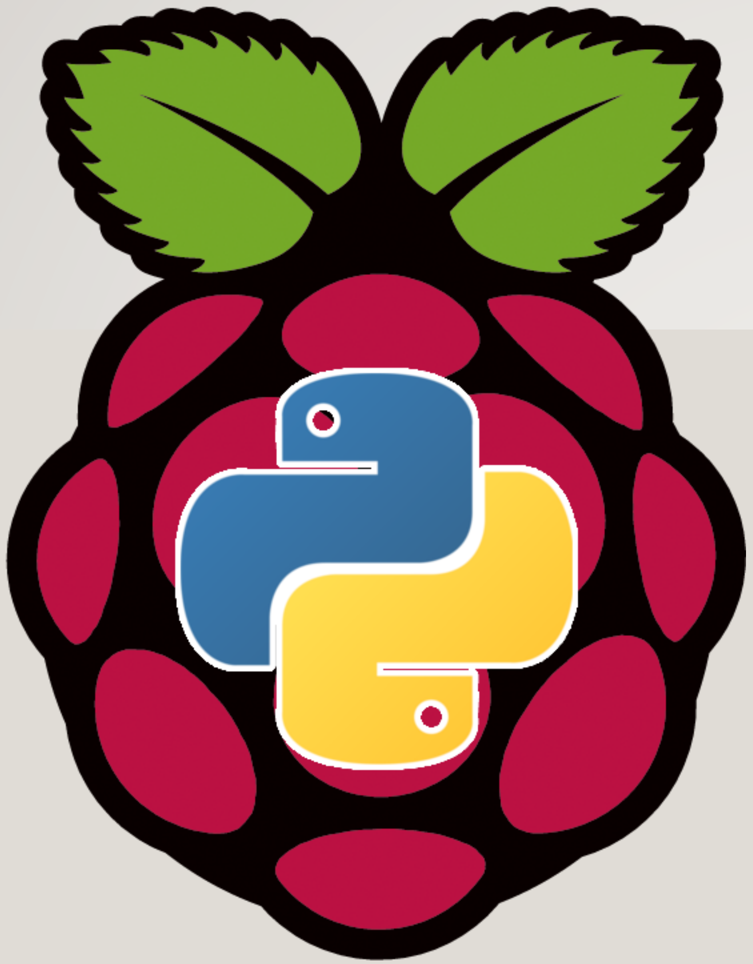
0%

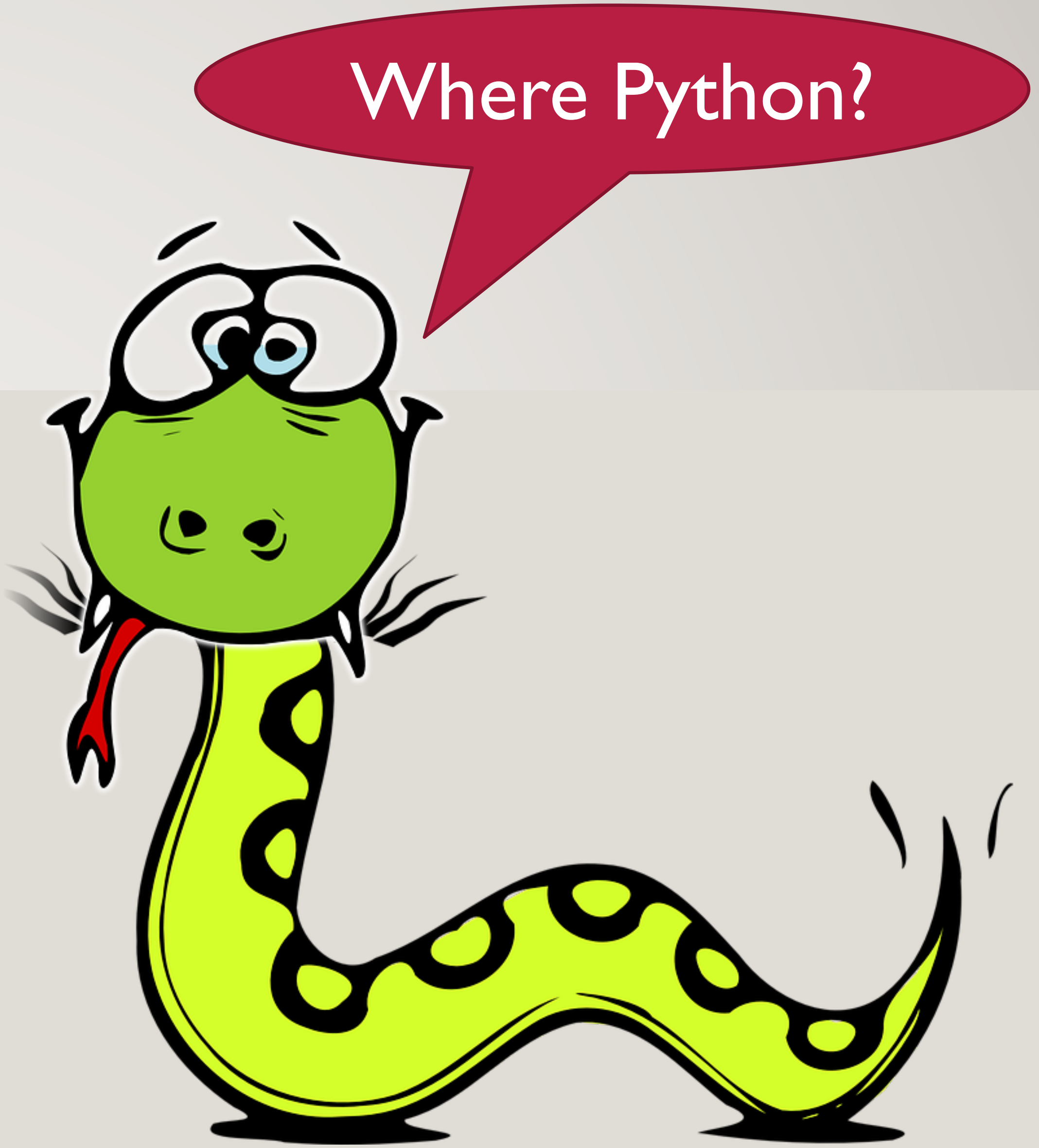
Desktop Applications



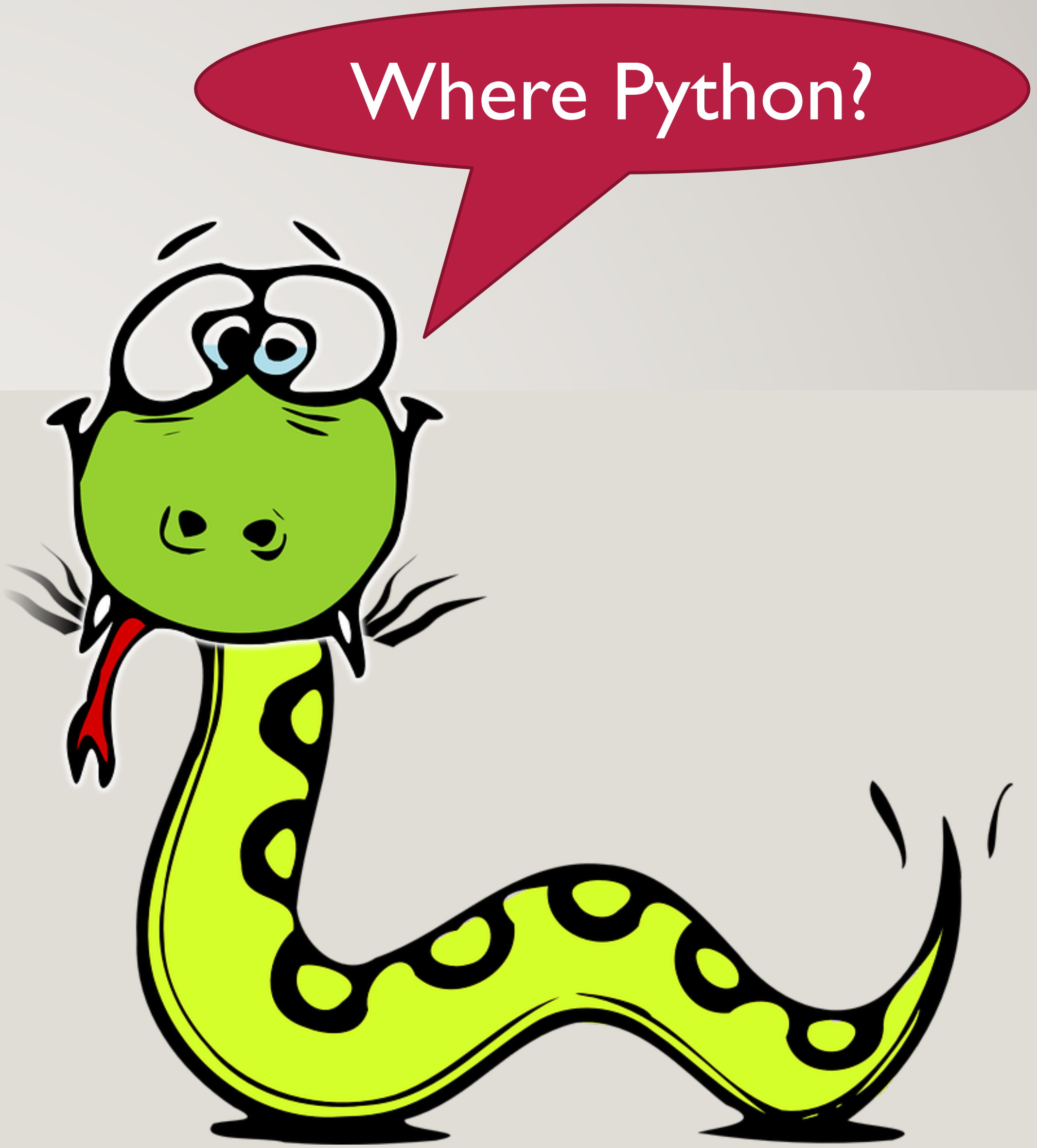
Games




IoT



Microsoft
SQL Server 2017


Where Python?

Where Python?
Not today, but maybe soon!

Getting Python

Windows - Use Chocolatey

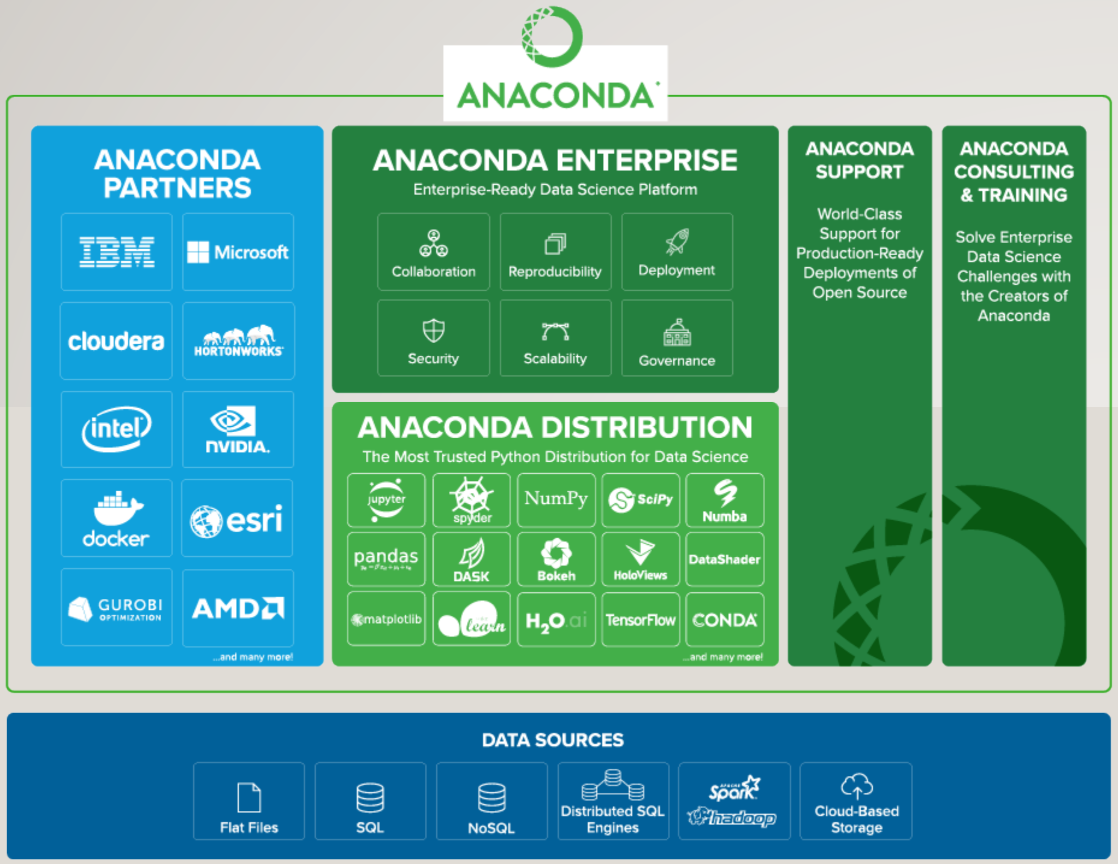
macOS - Use Homebrew

Do NOT use MacPorts!

Linux – It depends

Use the package manager for your distribution:
ie. apt – Debian/Ubuntu
yum - Fedora

@poweredbyaltnet



Python and R distribution tailored for data science

Over 200 open source packages, tools and utilities

Meet PEP-8

PEP – Python Enhancement Proposal

Suggestions for modifications and additions to the language

PEP-8 is a *recommended* style guide

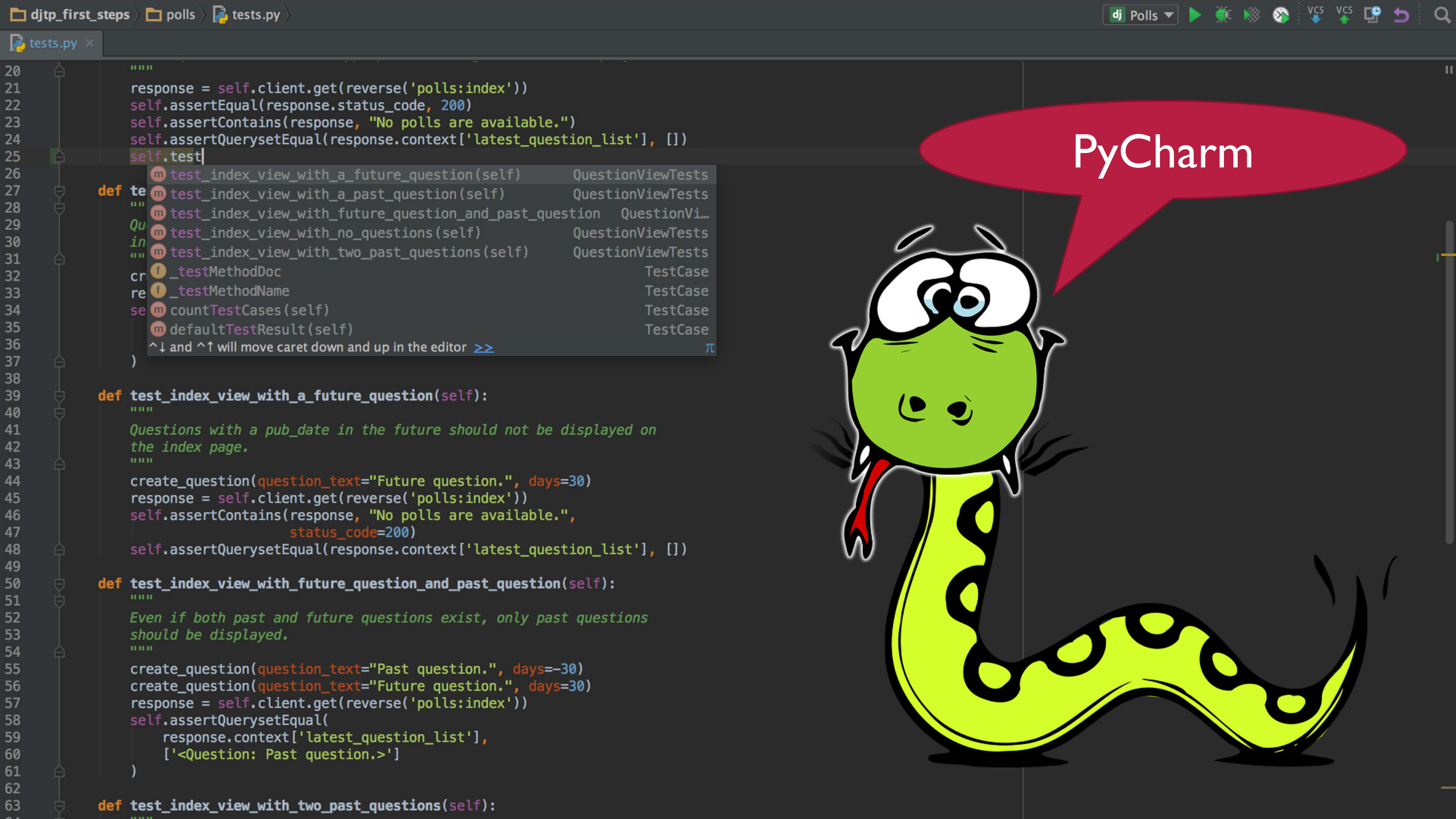
Many Pythonistas use at least part of it

Variable, function, method names use snake_case

Constants are upper case SNAKE_CASE

Class names are CamelCase

Lines are no more than 80 characters long

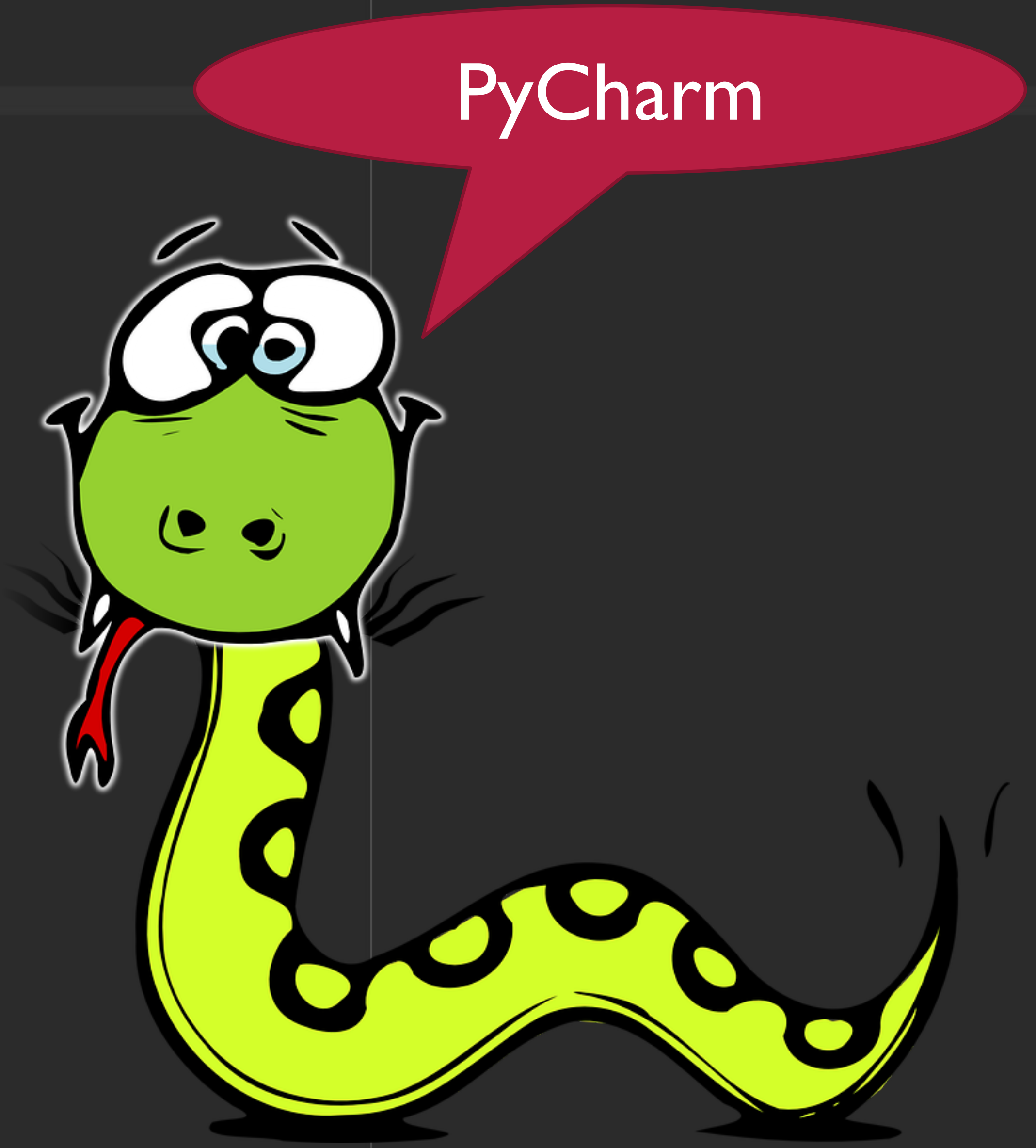


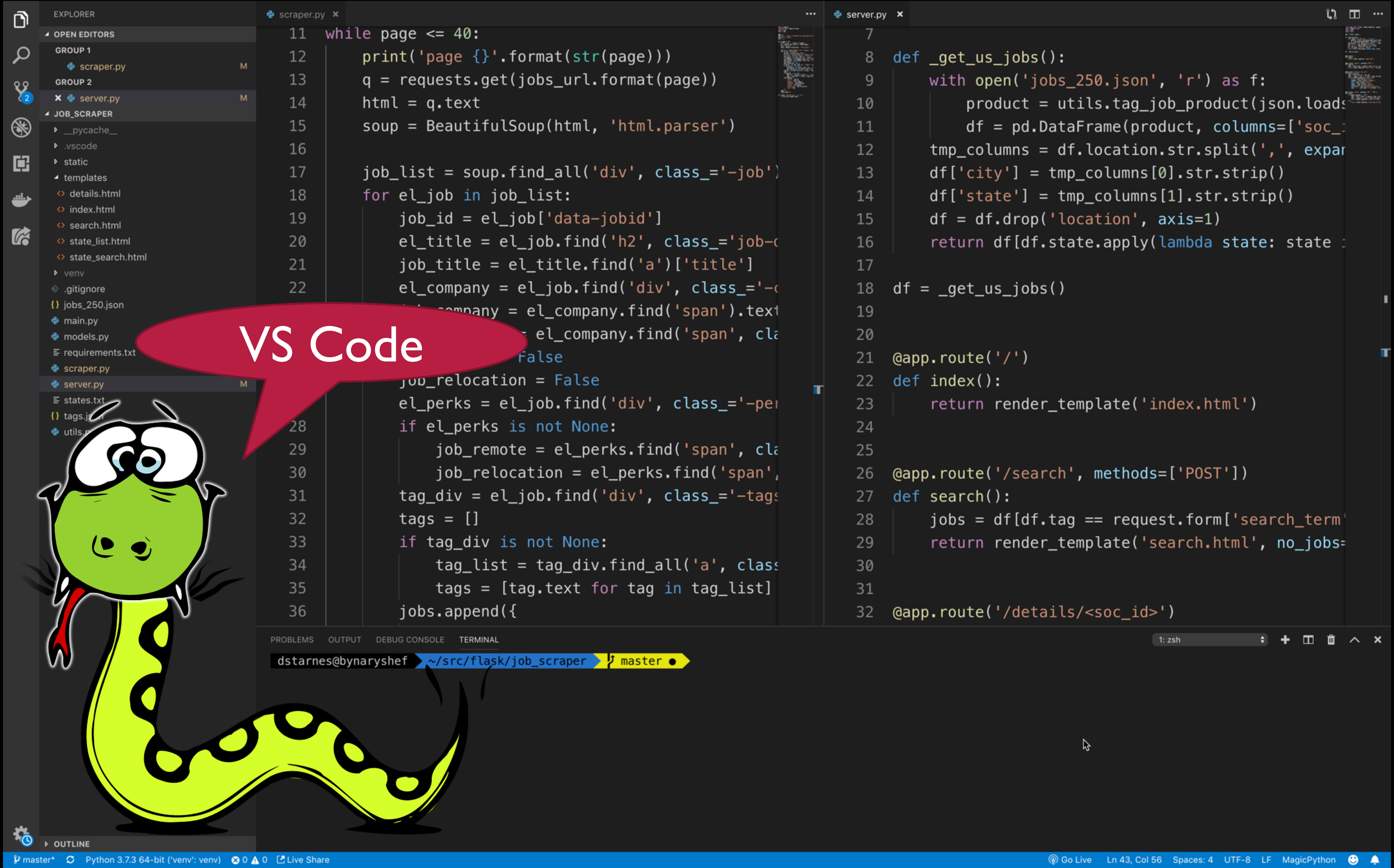
djtp_first_steps › polls › tests.py

tests.py ×

```
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])
        self.test

    def te
        """
        Qu
        in
        """
        cr
        re
        se



        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.",
                            status_code=200)
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
```

```
m test_index_view_with_a_future_question(self)          QuestionViewTests
m test_index_view_with_a_past_question(self)            QuestionViewTests
m test_index_view_with_future_question_and_past_question   QuestionVi…
m test_index_view_with_no_questions(self)               QuestionViewTests
m test_index_view_with_two_past_questions(self)         QuestionViewTests
f _testMethodDoc                                                 TestCase
f _testMethodName                                                TestCase
m countTestCases(self)                                           TestCase
m defaultTestResult(self)                                        TestCase
^↓ and ^↑ will move caret down and up in the editor >>                  π
```





EXPLORER

OPEN EDITORS
GROUP 1
scraper.py M
GROUP 2
server.py M
JOB_SCRAPER
__pycache__
.vscode
static
templates
details.html
index.html
search.html
state_list.html
state_search.html
venv
.gitignore
jobs_250.json
main.py
models.py
requirements.txt
scraper.py
server.py M
states.txt
tags.j
utils.p
OUTLINE

scraper.py

```
while page <= 40:
    print('page {}'.format(str(page)))
    q = requests.get(jobs_url.format(page))
    html = q.text
    soup = BeautifulSoup(html, 'html.parser')

    job_list = soup.find_all('div', class_='-job')
    for el_job in job_list:
        job_id = el_job['data-jobid']
        el_title = el_job.find('h2', class_='job-c
        job_title = el_title.find('a')['title']
        el_company = el_job.find('div', class_='-c
          ompany = el_company.find('span').text
                 = el_company.find('span', cla
                 False
        job_relocation = False
        el_perks = el_job.find('div', class_='-per
        if el_perks is not None:
            job_remote = el_perks.find('span', cla
            job_relocation = el_perks.find('span',
        tag_div = el_job.find('div', class_='-tags
        tags = []
        if tag_div is not None:
            tag_list = tag_div.find_all('a', class
            tags = [tag.text for tag in tag_list]
        jobs.append({
```

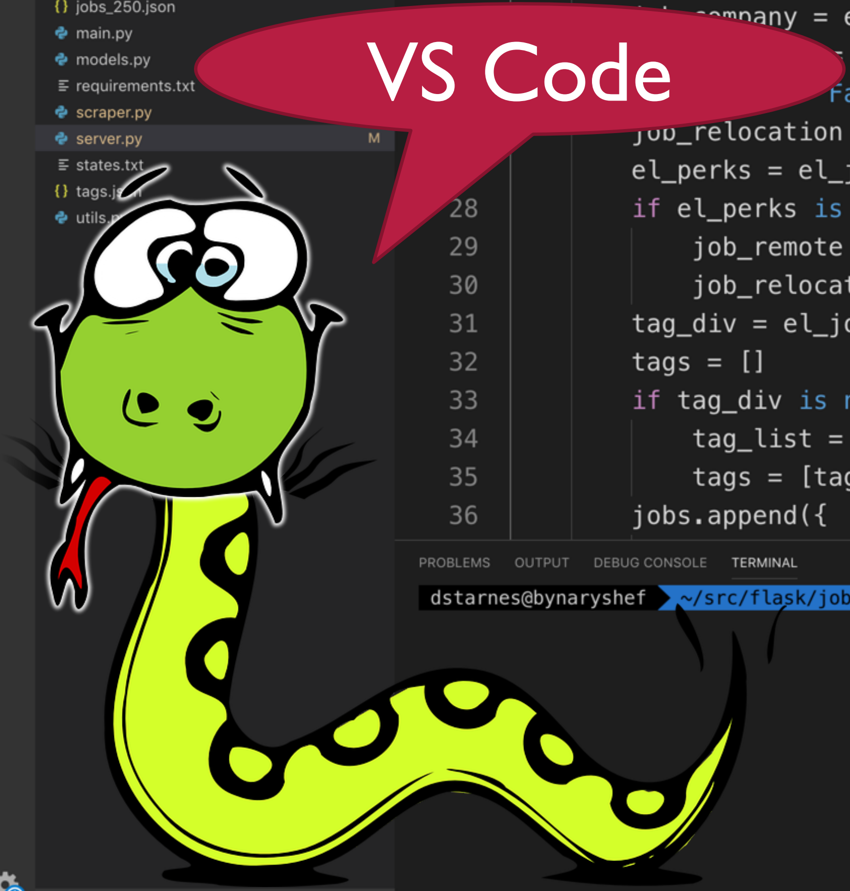
server.py

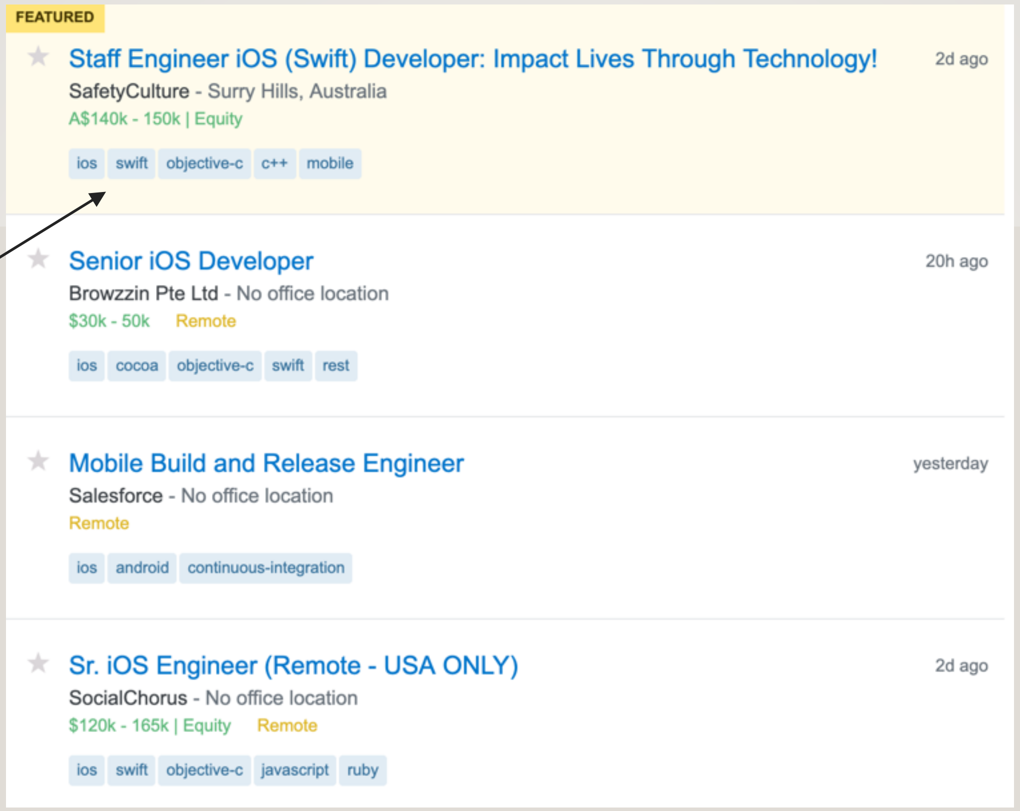
```

def _get_us_jobs():
    with open('jobs_250.json', 'r') as f:
        product = utils.tag_job_product(json.loads
        df = pd.DataFrame(product, columns=['soc_
    tmp_columns = df.location.str.split(',', expar
    df['city'] = tmp_columns[0].str.strip()
    df['state'] = tmp_columns[1].str.strip()
    df = df.drop('location', axis=1)
    return df[df.state.apply(lambda state: state

df = _get_us_jobs()


@app.route('/')
def index():
    return render_template('index.html')


@app.route('/search', methods=['POST'])
def search():
    jobs = df[df.tag == request.form['search_term'
    return render_template('search.html', no_jobs=


@app.route('/details/<soc_id>')
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: zsh

```
dstarnes@bynaryshef  ~/src/flask/job_scraper  master ●
```

master* Python 3.7.3 64-bit ('venv': venv) ⊗ 0 ⚠ 0 Live Share Go Live Ln 43, Col 56 Spaces: 4 UTF-8 LF MagicPython





FEATURED

Staff Engineer iOS (Swift) Developer: Impact Lives Through Technology! 2d ago
SafetyCulture - Surry Hills, Australia
A$140k - 150k | Equity
ios swift objective-c c++ mobile

Senior iOS Developer 20h ago
Browzzin Pte Ltd - No office location
$30k - 50k Remote
ios cocoa objective-c swift rest

Mobile Build and Release Engineer yesterday
Salesforce - No office location
Remote
ios android continuous-integration

Sr. iOS Engineer (Remote - USA ONLY) 2d ago
SocialChorus - No office location
$120k - 165k | Equity Remote
ios swift objective-c javascript ruby

https://stackoverflow.com/jobs

@poweredbyaltnet

Determine the most popular tags on the StackOverflow jobs site

REQUIREMENTS

- Get the HTML content from the StackOverflow site
- Parse the HTML and find the elements for tags
- Count the number of times each tag is used

GET THE HTML DATA FROM THE STACKOVERFLOW SITE

- requests
- Open source community package
- Simplifies making HTTP requests

```
$ pip install requests
```

```
import requests
```

PARSE THE HTML AND FIND THE ELEMENTS FOR TAGS

- beautifulsoup4
- Open source community package
- Makes navigating HTML content easier

```
$ pip install beautifulsoup4
```

```python
import requests
from bs4 import BeautifulSoup
```

Import the *entire* requests module, but no specific members
Member access requires module prefix

Import a specific member from the module
No prefix required

```
import requests
from bs4 import BeautifulSoup

jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
```

Strings may be double or single quoted
Single quoted recommended

Variable declaration is just var = value
No type specification
Must initialize variable

```
import requests
from bs4 import BeautifulSoup

jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
q = requests.get(jobs_url)
```

`get()` is a function inside of `requests`
Prefixed with module name
Sends HTTP request to the url passed
Returns HTTP response

```
import requests
from bs4 import BeautifulSoup

jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
q = requests.get(jobs_url)
html = q.text
```

The response contains data about the status code, headers, etc.
For time I am assuming the response has a 200 code
The content (in this case HTML markup) is in the text field

```
import requests
from bs4 import BeautifulSoup

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')
```

The HTML text is parsed by an instance of `BeautifulSoup`
The parser used is an HTML parser

Instantiating an object does not require a new keyword
The class name alone, and parameters in parentheses
This is an initializer, *not* a constructor (more later)

@poweredbyaltnet

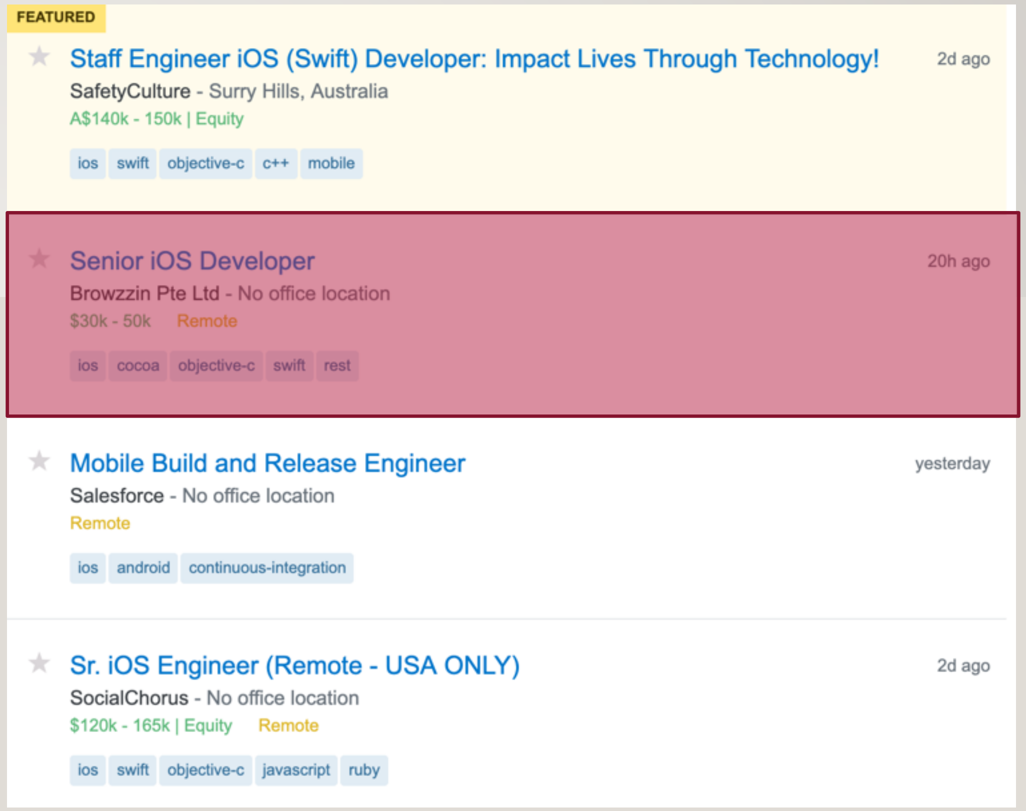
```
import requests
from bs4 import BeautifulSoup

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')

job_list = soup.find_all('div', class_='-job')
```

Each job post is wrapped in a <div> tag with a CSS class of '-job'

The `find_all` method will return a list of all <div> tags with a CSS class of '-job' in the parsed HTML

`class_` is a *keyword argument*

The trailing underscore removes ambiguity with the reserved class word



FEATURED

Staff Engineer iOS (Swift) Developer: Impact Lives Through Technology! 2d ago
SafetyCulture - Surry Hills, Australia
A$140k - 150k | Equity
ios swift objective-c c++ mobile

Senior iOS Developer 20h ago
Browzzin Pte Ltd - No office location
$30k - 50k Remote
ios cocoa objective-c swift rest

Mobile Build and Release Engineer yesterday
Salesforce - No office location
Remote
ios android continuous-integration

Sr. iOS Engineer (Remote - USA ONLY) 2d ago
SocialChorus - No office location
$120k - 165k | Equity Remote
ios swift objective-c javascript ruby

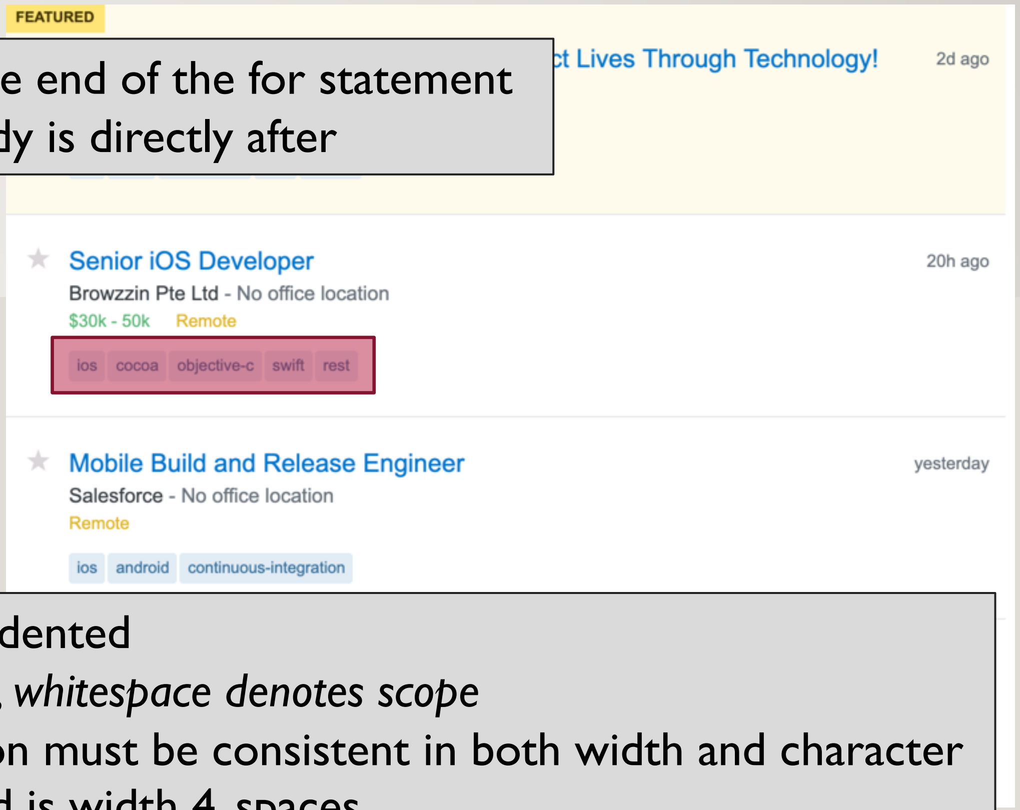
A list in Python is similar to an array in JavaScript

Ordered, linear collection of valid Python values of heterogenous type

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup

jobs_url = ‘https://stackoverflow.com/jobs?sort=p
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, ‘html.parser’)

job_list = soup.find_all(‘div’, class_=‘-job’)
for el_job in job_list:
    tag_div = el_job.find(‘div’, class_=‘-tags’)
```

The colon at the end of the for statement means the body is directly after


FEATURED

…ct Lives Through Technology! 2d ago

Senior iOS Developer 20h ago
Browzzin Pte Ltd - No office location
$30k - 50k Remote
ios cocoa objective-c swift rest

Mobile Build and Release Engineer yesterday
Salesforce - No office location
Remote
ios android continuous-integration

Python has no for-next loop like C
For loops iterate over a source

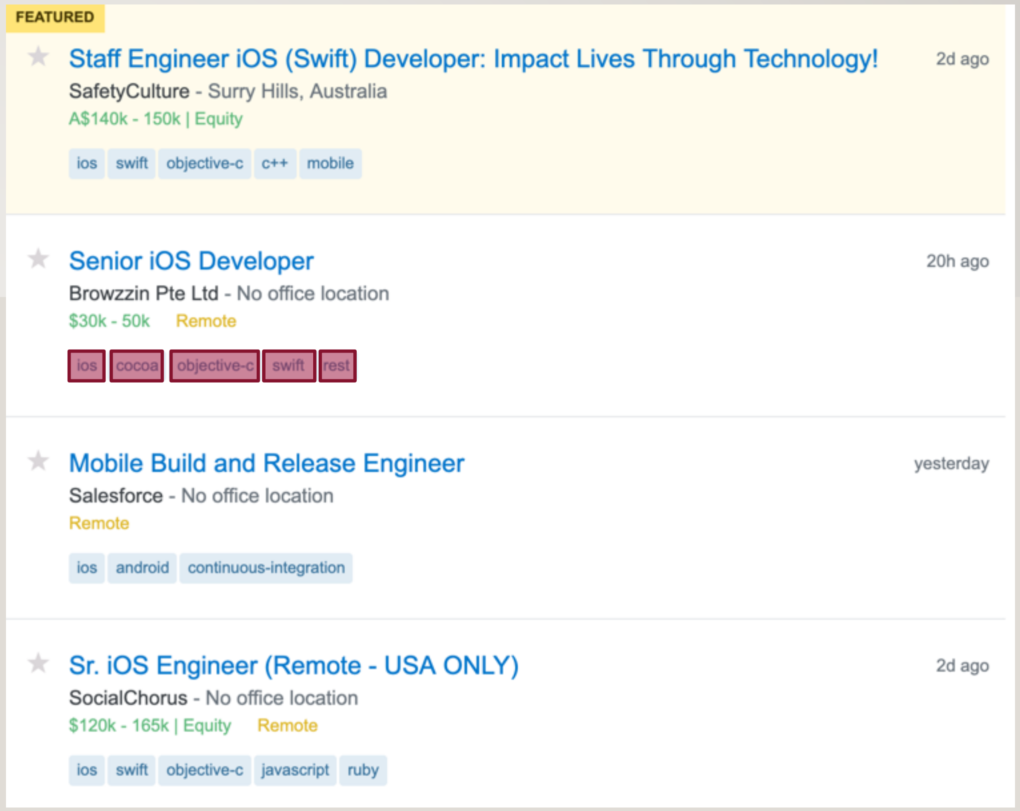
The body is indented
This is required, whitespace denotes scope
The indentation must be consistent in both width and character
Recommended is width 4, spaces

The tags are enclosed in a <div> with a CSS class of ‘-tags’
The `find()` method will return the first (in this case only) <div> with a CSS class of ‘-tags’
Only descendant of the `el_job` tag will be searched

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')

job_list = soup.find_all('div', class_='-job')
for el_job in job_list:
    tag_div = el_job.find('div', class_='-tags')
    tag_list = tag_div.find_all('a', class_='post-tag')
    for tag in tag_list:
        print(tag.text)
```

Each tag is an <a> with a CSS class of 'post-tag'
The tag itself is in the `text` field


FEATURED
Staff Engineer iOS (Swift) Developer: Impact Lives Through Technology! 2d ago
SafetyCulture - Surry Hills, Australia
A$140k - 150k | Equity
ios swift objective-c c++ mobile

Senior iOS Developer 20h ago
Browzzin Pte Ltd - No office location
$30k - 50k Remote
ios cocoa objective-c swift rest

Mobile Build and Release Engineer yesterday
Salesforce - No office location
Remote
ios android continuous-integration

Sr. iOS Engineer (Remote - USA ONLY) 2d ago
SocialChorus - No office location
$120k - 165k | Equity Remote
ios swift objective-c javascript ruby

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')

job_list = soup.find_all('div', class_='-job')
for el_job in job_list:
    tag_div = el_job.find('div', class_='-tags')
    if tag_div is not None:
        tag_list = tag_div.find_all('a', class_='post-tag')
        for tag in tag_list:
            print(tag.text)
```

Jobs are not required to have tags
A job without tags will omit the <div> with CSS class of '-tags'
In this case, the call to `find()` will return `None`
None is the null type in Python

If `tag_div` is None the call to `find_all()` will crash
Check that `tag_div` is not None
If statements are termintated with a colon
and the body is indented
Instead of != operator, Python is more readable

COUNT THE NUMBER OF TIMES EACH TAG IS USED

- Counter type in the collections package
- collections is part of the Python Standard Library
- Distributed with Python

```
import requests
from bs4 import BeautifulSoup
from collections import Counter

jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, ‘html.parser’)
tag_counter = Counter()

job_list = soup.find_all(‘div’, class_=‘-job’)
for el_job in job_list:
    tag_div = el_job.find(‘div’, class_=‘-tags’)
    if tag_div is not None:
        tag_list = tag_div.find_all(‘a’, class_=‘post-tag’)
        for tag in tag_list:
            tag_counter[tag.text] += 1
```

Explicitly import the `Counter` class
No module prefix needed

Create a new instance of `Counter`

`Counter` is a dictionary-like type
A dictionary is a collection of key-value pairs
The `Counter` tracks frequencies, tags in this case
Every non-existent key (tag) has a default value of 0
Accessing a non-existent key will not crash

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup
from collections import Counter

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')
tag_counter = Counter()

job_list = soup.find_all('div', class_='-job')
for el_job in job_list:
    tag_div = el_job.find('div', class_='-tags')
    if tag_div is not None:
        tag_list = tag_div.find_all('a', class_='post-tag')
        for tag in tag_list:
            tag_counter[tag.text] += 1

for item in tag_counter.most_common(25):
    print('{}, {} jobs'.format(item[0], str(item[1]))
```

The `most_common()` method will return the tags and counts in descending order
The number of tags returned is limited by an optional argument
The tags are a list of tuples
A tuple is similar to a list, except it is fixed length and immutable

Format strings use curly braces as placeholders
Tuple values can be accessed by position
The first value is the tag, the count is second
Explicitly cast the count (which is an integer) to a string

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup
from collections import Counter

jobs_url = 'https://stackoverflow.com/jobs?sort=p'
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, 'html.parser')
tag_counter = Counter()

job_list = soup.find_all('div', class_='-job')
for el_job in job_list:
    tag_div = el_job.find('div', class_='-tags')
    if tag_div is not None:
        tag_list = tag_div.find_all('a', class_='post-tag')
        for tag in tag_list:
            tag_counter[tag.text] += 1

for (index, (tag, count)) in enumerate(tag_counter.most_common(25)):
    print('Tag: {}, {} jobs'.format(str(index + 1), tag, str(count))
```

To display the relative position of the tags, the `enumerate()` function will return a list of tuples containing the index of an item and the associated item

Tuples can be destructured, even nest tuples

The index is zero-based

@poweredbyaltnet

```
import requests
from bs4 import BeautifulSoup
from collections import Counter

jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
q = requests.get(jobs_url)
html = q.text
soup = BeautifulSoup(html, ‘html.parser’)
tag_counter = Counter()

job_list = soup.find_all(‘div’, class_=‘-job’)
for el_job in job_list:
    tag_div = el_job.find(‘div’, class_=‘-tags’)
    if tag_div is not None:
        tag_list = tag_div.find_all(‘a’, class_=‘post-tag’)
        for tag in tag_list:
            tag_counter[tag.text] += 1

for (index, (tag, count)) in enumerate(tag_counter.most_common(25)):
    print(‘Tag: {}, {} job{}’.format(str(index + 1), tag, str(count), ‘s’ if count > 1 else ‘’)
```

Use the ternary if statement to handle the plural form

```
import requests
from bs4 import BeautifulSoup
from collections import Counter

page = 1
jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
jobs_url += ‘&pg={}’
tag_counter = Counter()

while page <= 40:
    q = requests.get(jobs_url.format(str(page)))
    html = q.text
    soup = BeautifulSoup(html, ‘html.parser’)

    job_list = soup.find_all(‘div’, class_=‘-job’)
    for el_job in job_list:
        tag_div = el_job.find(‘div’, class_=‘-tags’)
        if tag_div is not None:
            tag_list = tag_div.find_all(‘a’, class_=‘post-tag’)
            for tag in tag_list:
                tag_counter[tag.text] += 1
    page += 1

for (index, (tag, count)) in enumerate(tag_counter.most_common(25)):
    print(‘Tag {}: {}, {} job{}’.format(str(index + 1), tag, str(count), ‘s’ if count > 1 else ‘’)
```

Just 25 jobs is not enough for a representative sample

Keep parsing jobs until we have 40 pages worth
No parentheses around the condition
Also applies to if and for

```
$ python main.py

Tag 1: java, 272 jobs
Tag 2: javascript, 184 jobs
Tag 3: python, 160 jobs
Tag 4: sql, 121 jobs
Tag 5: reactjs, 114 jobs
Tag 6: c#, 105 jobs
Tag 7: amazon-web-services, 91 jobs
Tag 8: .net, 83 jobs
Tag 9: c++, 74 jobs
Tag 10: sysadmin, 73 jobs
Tag 11: cloud, 58 jobs
Tag 12: linux, 58 jobs
Tag 13: agile, 51 jobs
Tag 14: spring, 50 jobs
Tag 15: php, 49 jobs
Tag 16: node.js, 46 jobs
Tag 17: java-ee, 40 jobs
Tag 18: css, 38 jobs
Tag 19: docker, 38 jobs
Tag 20: web-services, 38 jobs
Tag 21: angularjs, 37 jobs
Tag 22: mysql, 35 jobs
Tag 23: testing, 33 jobs
Tag 24: ios, 33 jobs
Tag 25: windows, 33 jobs
```

```
import requests
from bs4 import BeautifulSoup
from collections import Counter
import json

page = 1
jobs_url = ‘https://stackoverflow.com/jobs?sort=p’
jobs_url += ‘&pg={}’
tag_counter = Counter()

while page <= 40:
    q = requests.get(jobs_url.format(str(page)))
    html = q.text
    soup = BeautifulSoup(html, ‘html.parser’)
```

```
        tag_div = el_job.find(‘div’, class_=‘-tags’)
        if tag_div is not None:
            tag_list = tag_div.find_all(‘a’, class_=‘post-tag’)
            for tag in tag_list:
                tag_counter[tag.text] += 1
    page += 1

for (index, (tag, count)) in enumerate(tag_counter.most_common(25)):
    print(‘Tag: {}, {} job{}’.format(str(index + 1), tag, str(count), ‘s’ if count > 1 else ‘’)

with open(‘tags.json’, ‘w’) as f:
    f.write(json.dumps(tag_counter))
```

StackOverflow would not appreciate us scraping 40 pages every time the application is run (they will rate limit you!) so we will persist the counts to a text file in JSON

Support for JSON is included in the Python standard library

The built-in `open()` function will return a file handle
The ‘w’ will open or create a file in write mode

Whatever is open, must eventually be closed
The `close()` method on a file handle will close the file
When used in a `with` statement, the file will automatically be closed when the body exits

The `write()` method will append the JSON string to the file.

The `dumps()` function will convert the dictionary into a JSON string

```
$ pip install ipython
```

IPython is an 'enhanced Python interpreter' that adds functionality to the default Python interactive REPL

It adds features like tab completion, automatic indentation, macros, and shell access

IPython is distributed as a Python package, and thus can be installed with pip

```
$ pip install ipython

$ ipython
$ pip install ipython

$ ipython

In [1] import json
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
```

This is as close as you can get to declaring a variable without initializing it

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
```

`loads()` will covert a JSON string to a Python dictionary/list
`join()` will concatenate the elements of a list using a delimiter

@poweredbyaltnet

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open('tags.json', 'r') as f:
           tags = json.loads(''.join(f.readlines()))
In [4] tags
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
In [4] tags
Out[4]:
{'java': 272,
 'python': 160,
 'sql': 121,
 'c': 32,
 'networking': 13,
 'testing': 33,
 'automated-tests': 14,
 'integration-testing': 2,
 'load-testing': 2,
 'hadoop': 10,
 'apache-spark': 9,
...
}
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
Out[7]
[('java', 272),
 ('javascript', 184),
 ('python', 160),
 ('sql', 121),
 ('reactjs', 114),
 ('c#', 105),
 ('amazon-web-services', 91),
 ('.net', 83),
 ('c++', 73),
 ('sysadmin', 73),
 ('cloud', 58),
...
]
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
            tags = json.loads(‘’.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
In [8] tag_counter[‘python’]
```

Again, `Counter` is a dictionary-like object

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open('tags.json', 'r') as f:
           tags = json.loads(''.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
In [8] tag_counter['python']
Out[8] 160
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
In [8] tag_counter[‘python’]
Out[8] 160
In [9] tag_counter[‘bogus’]
```

```
$ pip install ipython

$ ipython

In [1] import json
In [2] tags = None
In [3] with open(‘tags.json’, ‘r’) as f:
           tags = json.loads(‘’.join(f.readlines()))
In [4] tags
In [5] from collections import Counter
In [6] tag_counter = Counter(tags)
In [7] tag_counter.most_common(25)
In [8] tag_counter[‘python’]
Out[8] 160
In [9] tag_counter[‘bogus’]
Out[9] 0
```

Accessing a non-existent key will return 0

```
class Job(object):
```

To declare a class, use the `class` keyword
Followed by the name of the class in CamelCase
Followed by the parent class in parentheses (use object if there is no parent class)
End the statement with a colon

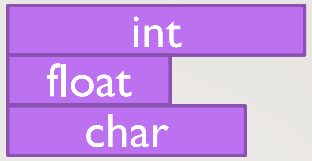
```
class Job(object):
     def __init__(self, id, title, company, location, tags=[], remote=False, relocation=False):
          self.id = id
          self.title = title
          self.company = company
          self.location = location
          self.tags = tags
          self.remote = remote
          self.relocation = relocation
```

The initializer is called during the creation of an instance of the class
Methods (and function) are preceded by the `def` keyword
The first parameter of an instance method, is the instance of the class
Parameters can have default values

The name of the initializer will always be `__init__`
This is pronounced 'dunder init dunder', or just 'dunder init'
'Dunder methods' are always preceded by two underscores
'Dunder methods' are reserved for use by Python
Never create your own dunder methods
(even though some prominent Pythonistas have gotten away with it)

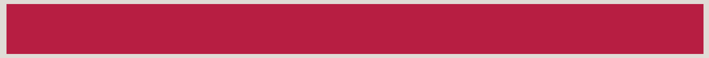
AN INITIALIZER IS NOT A CONSTRUCTOR

```
typedef struct foo {
    int i;
    float f;
    char c;
}Foo;
```

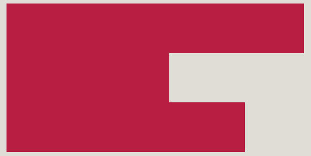


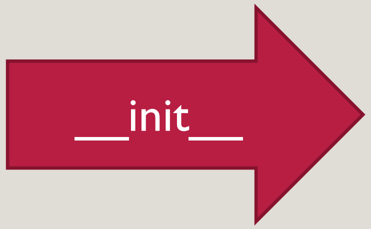
ALLOCATION

```
void *f = malloc(sizeof(Foo));
```

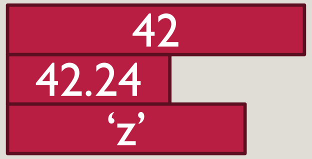


```
Foo *foo = (Foo *)f;
```





```
foo->i = 42;
foo->f = 42.24;
foo->c = ‘z’;
```



INITIALIZATION

```python
class Job(object):
    def __init__(self, id, title, company, location, tags=[], remote=False, relocation=False):
        self.id = id
        self.title = title
        self.company = company
        self.location = location
        self.tags = tags
        self.remote = remote
        self.relocation = relocation

    def __repr__(self):
        return '<Job {}>'.format(self.title)
```

`__repr__` is another dunder method
Returns the string representation of the instance
<__main__.Anything at 0x22fa4c9dc88> vs. <Job: Senior Fullstack Developer>

```
class Job(object):
    def __init__(self, id, title, company, location, tags=[], remote=False, relocation=False):
        self.id = id
        self.title = title
        self.company = company
        self.location = location
        self.tags = tags
        self.remote = remote
        self.relocation = relocation

    def __repr__(self):
        return '<Job {}>'.format(self.title)

    def add_tag(self, tag):
        """
        Adds ``tag`` if it does not already exist
        """
        if tag not in self.tags:
            self.tags.append(tag)

    def remove_tag(self, tag):
        """
        Removes ``tag`` if it already exists
        """
        if tag in self.tags:
            self.tags.remove(tag)
```

Instance methods also have the class instance as an implied first parameter
The triple quoted text is docstring (PEP-257)
Docstring precedes the first statement of a class, method or function
Used in IPython as part of the integrated help

```
$ ipython
```

@poweredbyaltnet

```
$ ipython

In [1] job = Job( ... )
```

```
$ ipython

In [1] job = Job( ... )
In [2] job.add_tag?
```

Integrated help in IPython is accessed by appending a '?' to a Python class/function/method

```
$ ipython

In [1] job = Job( ... )
In [2] job.add_tag?
Signature: job.add_tag(tag)
Docstring: Adds ``tag`` if it does not already exist
File:      c:\users\dougl\<ipython-input-6-99808ce3818d>
Type:      method
```

@poweredbyaltnet

```
class Job(object):
    def __init__(self, id, title, company, location, tags=[], remote=False, relocation=False):
        self.id = id
        self.title = title
        self.company = company
        self.location = location
        self.tags = tags
        self.remote = remote
        self.relocation = relocation

    def __repr__(self):
        return '<Job {}>'.format(self.title)

    def add_tag(self, tag):
        """
        Adds ``tag`` if it does not already exist
        """
        if tag not in self.tags:
            self.tags.append(tag)

    def remove_tag(self, tag):
        """
        Removes ``tag`` if it already exists
        """
        if tag in self.tags:
            self.tags.remove(tag)

    @classmethod
    def from_json(cls, json_str):
        json_data = json.loads(json_str)
        return cls(json_data['soc_id'], json_data['title'], json_data['company'], json_data['location'], json_data['tags’],
           json_data['remote'], json_data['relocation'])
```

`@classmethod` is a decorator

Decorators are metadata (ie. attributes in .NET, annotations in Java)

A classmethod is a method that is called on a class, instead of a class instance

The first parameter is the class itself (Job in this case)

```
$ ipython

In [1] import json
In [2] with open(‘data.json’, ‘r’) as f:
           json_str = ‘’.join(f.readlines())
           job = Job.from_json(json_str)
In [3] job.title
Out[3] Senior Fullstack Developer
```

@poweredbyaltnet

DATA WRANGLING

- The processing of mapping or transforming data from one format to another
- pandas
- Python package providing data structures and tools for data analysis
- Inspired by R
- DataFrame

```
$ pip install pandas
$ ipython
$ ipython

In [1] import pandas as pd
```

Pythonistas are lazy
This will import the package but use pd as the prefix

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
```

A DataFrame is a two dimensional structure similar to a R DataFrame

`job_data` for this example is a matrix/nested list
The default values for columns are integers

@poweredbyaltnet

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
 soc_id                           title company  location  remote  relocation                  tag
0  245603  Senior Java Software Developer   NAVIS  Bend, OR    True        True                 java
1  245603  Senior Java Software Developer   NAVIS  Bend, OR    True        True  amazon-web-services
2  245603  Senior Java Software Developer   NAVIS  Bend, OR    True        True               python
3  245603  Senior Java Software Developer   NAVIS  Bend, OR    True        True           postgresql
4  245603  Senior Java Software Developer   NAVIS  Bend, OR    True        True              reactjs
```

head() returns the first 5 (by default) rows

@poweredbyaltnet

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
0          True
1          True
2          True
3          True
4          True
5         False
6         False
7         False
8         False
9         False
```

The columns of the DataFrame are dynamically added as fields
A Series is basically an array with an index

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
0      245603                    Senior Java Software Developer  ...       True                    java
1      245603                    Senior Java Software Developer  ...       True   amazon-web-services
2      245603                    Senior Java Software Developer  ...       True                  python
3      245603                    Senior Java Software Developer  ...       True              postgresql
4      245603                    Senior Java Software Developer  ...       True                 reactjs
28     253986          Software Engineer-web/mobile/CMS back-end  ...      False                    java
29     253986          Software Engineer-web/mobile/CMS back-end  ...      False              javascript
30     253986          Software Engineer-web/mobile/CMS back-end  ...      False                 node.js
31     253986          Software Engineer-web/mobile/CMS back-end  ...      False                  python
32     253986          Software Engineer-web/mobile/CMS back-end  ...      False           microservices
```

A Series of booleans can filter a DataFrame

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
```

Access the values of the location column as strings
Split on the comma
Put the split values into new columns in a new DataFrame

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
```

New columns are added dynamically to a DataFrame using square brackets
`strip()` will remove whitespace from both ends of a string

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
In [9] df = df.drop(‘location’, axis=1)
```

The location column is no longer needed, so it can be removed
The axis specifies whether to drop rows or columns (0 – rows, 1 – columns)

@poweredbyaltnet

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
In [9] df = df.drop(‘location’, axis=1)
In[10] us_companies = df[df.state.apply(lambda state: state in utils.get_states())]
```

`get_states()` is a function I have written that returns a list of state abbreviations
`apply()` will call a given function, and pass each row in the state column
Python’s lambda syntax automatically returns the result of the body

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
In [9] df = df.drop(‘location’, axis=1)
In[10] us_companies = df[df.state.apply(lambda state: state in utils.get_states())]
In[11] us_companies.head()
 soc_id                           title company  remote  relocation                  tag  city state
0  245603  Senior Java Software Developer   NAVIS    True        True                 java  Bend    OR
1  245603  Senior Java Software Developer   NAVIS    True        True  amazon-web-services  Bend    OR
2  245603  Senior Java Software Developer   NAVIS    True        True               python  Bend    OR
3  245603  Senior Java Software Developer   NAVIS    True        True           postgresql  Bend    OR
4  245603  Senior Java Software Developer   NAVIS    True        True              reactjs  Bend    OR
```

@poweredbyaltnet

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
In [9] df = df.drop(‘location’, axis=1)
In[10] us_companies = df[df.state.apply(lambda state: state in utils.get_states())]
In[11] us_companies.head()
In[12] us_companies.state.unique()
array(['OR', 'NY', 'IL', 'TX', 'NC', 'GA', 'CA', 'MN', 'MI', 'PA', 'AZ',
       'FL', 'MD', 'UT', 'CO', 'DE', 'NJ', 'OH', 'WA', 'WI', 'MA', 'TN',
       'AR', 'VA', 'OK', 'CT', 'SC', 'IN', 'ID', 'MO', 'NH', 'NV', 'IA',
       'NE', 'VT', 'KY', 'ME', 'NM', 'RI'], dtype=object)
```

`unique()` will remove duplicate values

@poweredbyaltnet

```
$ ipython

In [1] import pandas as pd
In [2] df = pd.DataFrame(job_data, columns = [‘soc_id’, ’title’, ‘company’, ‘location’, ‘remote’, ‘relocation’, ‘tag’]
In [3] df.head()
In [4] df.remote == True
In [5] df[df.remote == True]
In [6] tmp_columns = df.location.str.split(‘,’, expand=True)
In [7] df[‘city’] = tmp_columns[0].str.strip()
In [8] df[‘state’] = tmp_columns[1].str.strip()
In [9] df = df.drop(‘location’, axis=1)
In[10] us_companies = df[df.state.apply(lambda state: state in utils.get_states())]
In[11] us_companies.head()
In[12] us_companies.state.unique()
In[13] set(us_companies.state.unique()).difference(set(utils.get_states()))
set()
```

A `set()` is like a tuple that disallows duplicates
If all values in the state column are in the list of state abbreviations, the difference between the two sets will be empty

@poweredbyaltnet

WEB APPLICATIONS

- Flask
- Microframework for building web applications with Python
- Like Python, it stays out of your way
- Unopinionated
- Other frameworks, like Django, work best for preconceived scenarios (ie. forms-over-data)

```
$ pip install flask
```

```
from flask import Flask
```

@poweredbyaltnet

```
from flask import Flask

app = Flask(__name__)
```

Flask will create a application which can be hosted by a WSGI server
`__name__` represents the current module

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World</h1>'
```

The route decorator will map URLs to handler functions
The return value of the handler function will be sent to the client in an HTTP response.

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')
```

The `render_template` function, by default, looks for HTML templates in the templates/ directory

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route(‘/’)
def index():
    return render_template(‘index.html’)

@app.route(‘/search’, methods=[‘POST’])
def search():
    jobs = df[df.tag == request.form[‘search_term’]]
    return render_template(‘search.html’
                           search_term=request.form[‘search_term’],
                           no_jobs=len(jobs),
                           jobs=[job[1] for job in jobs.iterrows()]
```

The methods keyword argument lists the HTTP verbs that will be handled

The keyword arguments to `render_template` are made available in the template itself

```
<div>
    <h3>I found {{ no_jobs }} jobs matching {{ search_term }}</h3>
</div>
<table>
    {% for job in jobs %}
        <tr>
            <td>
                <a href="/details/{{ job['soc_id'] }}">
                    {{ job['title'] }}
                </a>
            </td>
        </tr>
    {% endfor %}
</table>
```

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route(‘/’)
def index():
    return render_template(‘index.html’)

@app.route(‘/search’, methods=[‘POST’])
def search():
    jobs = df[df.tag == request.form[‘search_term’]]
    return render_template(‘search.html’
                           search_term=request.form[‘search_term’],
                           no_jobs=len(jobs),
                           jobs=[job[1] for job in jobs.iterrows()]
```

A list comprehension is a shortcut for a for loop or the `map()` function

```
<div>
    <h3>I found {{ no_jobs }} jobs matching {{ search_term
}}</h3>
</div>
<table>
    {% for job in jobs %}
        <tr>
            <td>
                <a href="/details/{{ job['soc_id'] }}">
                    {{ job['title'] }}
                </a>
            </td>
        </tr>
    {% endfor %}
</table>
```

@poweredbyaltnet

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route(‘/’)
def index():
    return render_template(‘index.html’)

@app.route(‘/search’, methods=[‘POST’])
def search():
    jobs = df[df.tag == request.form[‘search_term’]]
    return render_template(‘search.html’
                           search_term=request.form[‘search_term’],
                           no_jobs=len(jobs),
                           jobs=[job[1] for job in jobs.iterrows()]

@app.route(‘/details/<soc_id>’)
def details(soc_id):
    jobs = df[df.soc_id == soc_id]
    # omitted for space
    # by the way, this is a comment
    return render_template(‘details.html’)
```

Rules surrounded by angle brackets map to parameters passed to the handler function

```
$ export FLASK_APP=server.py
$ export FLASK_APP=server.py
$ export FLASK_DEBUG=1
$ export FLASK_APP=server.py
$ export FLASK_DEBUG=1
$ flask run
$ export FLASK_APP=server.py
$ export FLASK_DEBUG=1
$ flask run
$ flask shell
```

What's 'Missing'?

- Truncated set of data types (ie. only 3 numeric types)
- Increment (++) and decrement (--) operators
- Constants
- Switch statement
- Generics
- Interfaces
- Single inheritance

@poweredbyaltnet

Recall the 3 pillars of OOP

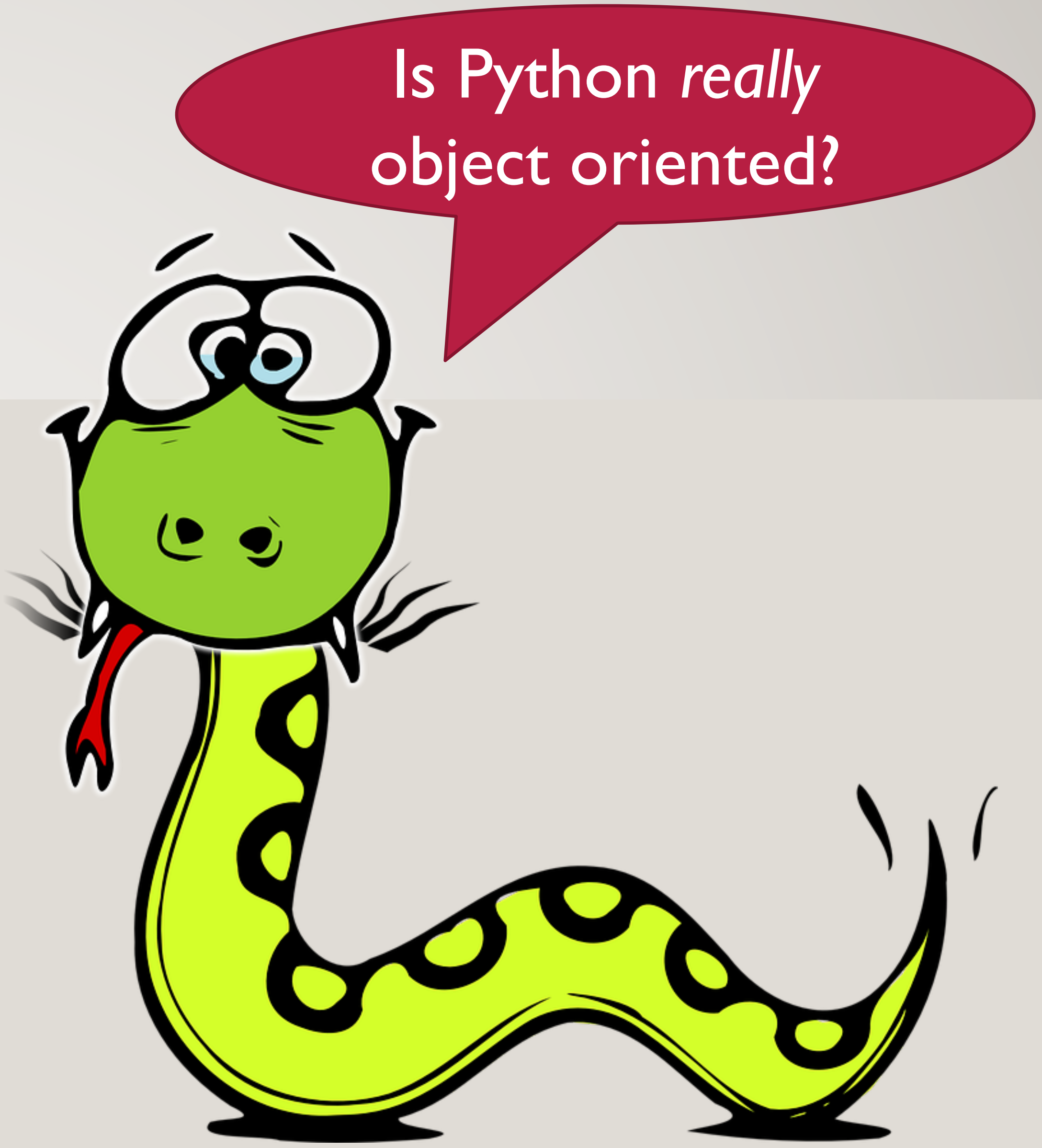
Polymorphism

Inheritance

Encapsulation

Python has no access modifiers

All class members in Python are effectively public



```
class UserType(object):
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def _shout_about_self(self):
        return ‘{} {}’.format(self.bar.upper(), self.bar.upper())

    def yell_about_self(self):
        return self._shout_about_self()
```

@poweredbyaltnet

```
class UserType(object):
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def _shout_about_self(self):
        return ‘{} {}’.format(self.bar.upper(), self.bar.upper())

    def yell_about_self(self):
        return self._shout_about_self()
```

```
$ ipython
```

```
class UserType(object):
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def _shout_about_self(self):
        return ‘{} {}’.format(self.bar.upper(), self.bar.upper())

    def yell_about_self(self):
        return self._shout_about_self()

$ ipython

In [1] ut = UserType(‘foo’, ‘bar’)
```

@poweredbyaltnet

```
class UserType(object):
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def _shout_about_self(self):
        return '{} {}'.format(self.bar.upper(), self.bar.upper())

    def yell_about_self(self):
        return self._shout_about_self()

$ ipython

In [1] ut = UserType('foo', 'bar')
In [2] ut.yell_about_self()
Out[2] 'FOO BAR'
```

@poweredbyaltnet

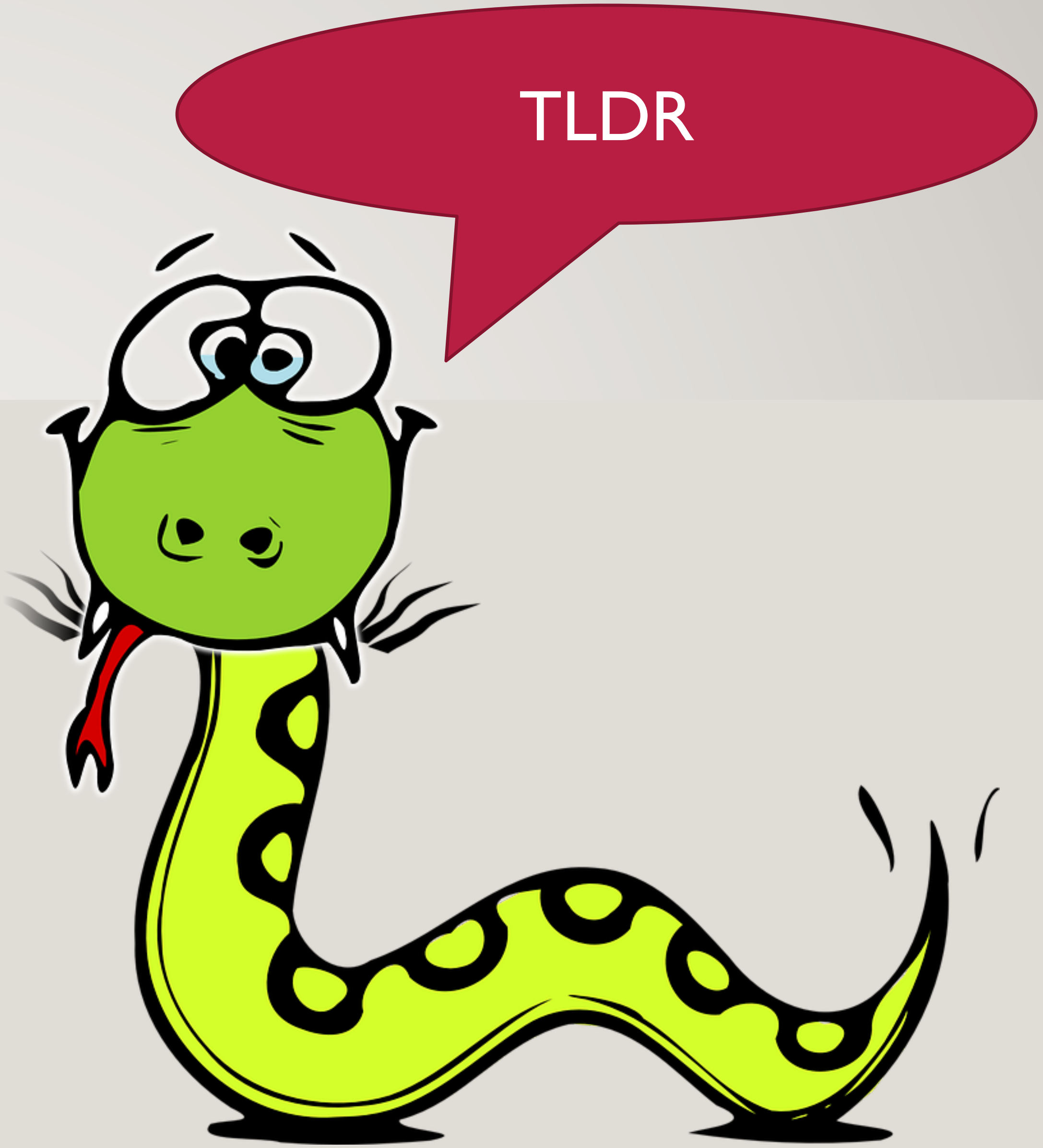
```
class UserType(object):
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    def _shout_about_self(self):
        return ‘{} {}’.format(self.bar.upper(), self.bar.upper())

    def yell_about_self(self):
        return self._shout_about_self()

$ ipython

In [1] ut = UserType(‘foo’, ‘bar’)
In [2] ut.yell_about_self()
Out[2] ‘FOO BAR’
In [3] ut._shout_about_self()
Out[3] ‘FOO BAR’
```

Don’t do this!
Private functions may only be accessed by the classes they belong to

You can't fix stupid

You can't protect people from themselves

Any attempt to accommodate these people will punish responsible developers



"We're all adults here"

The Zen of Python

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

```
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.
Namespaces are one honking great idea -- let's do more of
those!
```

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.
Namespaces are one honking great idea -- let's do more of
those!
```

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.
Namespaces are one honking great idea -- let's do more of
those!
```

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.
Namespaces are one honking great idea -- let's do more of
those!
```

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.
Namespaces are one honking great idea -- let's do more of
those!
```

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.
Namespaces are one honking great idea -- let's do more of
those!
```

THANK YOU

- douglas@douglasstarnes.com
- @poweredbyaltnet
- http://douglasstarnes.com