Python Bootcamp
Day 3

Brian Haugen
Daniel Lee
Garrett Walter

PEP
(Python Enhancement Proposals)

https://www.python.org/dev/peps/

Style Guide for Python Code (PEP 8)

Describes standard conventions for code style

Basically, install a linter and use it

```
> pip install flake8
> flake8 script.py
> flake8 project_dir
```

PEP8 Code Layout

- Indentation: 4 spaces per indentation level
- Tabs vs spaces: Use spaces
- Line length:
  - Code: 79 characters
  - Comments and docstrings: 72 characters

PEP8 Naming Conventions

Functions/variables: snake_case

```python
def some_function():
    long_variable_name = do_something()
```

Classes: CamelCase

```python
class SuperCoolClass:
```

Constants: ALL_CAPS

```python
MAX_K = 50
```

Package/modules: shortlower

```python
import tensorflow as tf
```

Docstrings (PEP 257)

A string literal as the first statement in a definition (always triple quotes)

Accessed using `func_name.__doc__`

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
```

Zen of Python

Key points of Zen of Python (PEP 20)

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Sparse is better than dense
- Readability counts

and 14 more…

See them all by doing `import this`

Simple is better than complex

DON'T

```
def store(measurements):
    import sqlalchemy
    import sqlalchemy.types as sqltypes

    db = sqlalchemy.create_engine('sqlite:///measurements.db')
    db.echo = False
    metadata = sqlalchemy.MetaData(db)
    table = sqlalchemy.Table('measurements', metadata,
        sqlalchemy.Column('id', sqltypes.Integer, primary_key=True),
        sqlalchemy.Column('weight', sqltypes.Float),
        sqlalchemy.Column('temperature', sqltypes.Float),
        sqlalchemy.Column('color', sqltypes.String(32)),
        )
    table.create(checkfirst=True)

    for measurement in measurements:
        i = table.insert()
        i.execute(**measurement)
```

DO

```
def store(measurements):
    import json
    with open('measurements.json', 'w') as f:
        f.write(json.dumps(measurements))
```

Sparse is better than dense

DON'T

```python
def process(response):
    selector = lxml.cssselect.CSSSelector('#main > div.text')
    lx = lxml.html.fromstring(response.body)
    title = lx.find('./head/title').text
    links = [a.attrib['href'] for a in lx.find('./a') if 'href' in a.attrib]
    for link in links:
        yield Request(url=link)
    divs = selector(lx)
    if divs: yield Item(utils.lx_to_text(divs[0]))
```

Sparse is better than dense

DO

```
def process(response):
    lx = lxml.html.fromstring(response.body)

    title = lx.find('./head/title').text

    links = [a.attrib['href'] for a in lx.find('./a') if 'href' in a.attrib]
    for link in links:
        yield Request(url=link)

    selector = lxml.cssselect.CSSSelector('#main > div.text')
    divs = selector(lx)
    if divs:
        bodytext = utils.lx_to_text(divs[0])
        yield Item(bodytext)
```

Readability counts

DON'T

```python
# A dictionary of families who live in each city
mydict = {
    'Midtown': ['Powell', 'Brantley', 'Young'],
    'Norcross': ['Montgomery'],
    'Ackworth': []
}


def a(dict):
    # For each city
    for p in dict:
        # If there are no families in the city
        if not mydict[p]:
            # Say that there are no families
            print('None.')
```

DO

```python
families_by_city = {
    'Midtown': ['Powell', 'Brantley', 'Young'],
    'Norcross': ['Montgomery'],
    'Ackworth': [],
}


def no_families(cities):
    for city in cities:
        if not len(families_by_city[city]):
            print(f'No families in {city}.')
```

https://realpython.com/python-comments-guide/

Classes

The Basics

```python
class SomeClass:
    # Constructor
    def __init__(self, param):
        self.instance_variable = param

    def method_with_no_params(self):
        return self.instance_variable * 2
```

A Little More Complex

```
class Vector:
    # Class attributes
    vectors_constructed = 0

    # Constructor
    def __init__(self, x, y):
        # Instance attributes
        self.x = x
        self.y = y
        Vector.vectors_constructed += 1

    def get_tuple(self):
        return (self.x, self.y)

    # Special (dunder) method
    def __add__(p):
        return Vector(self.x + p.x, self.y + p.y)
```

What can you do with dunder methods?

- Object representation
- Iteration
- Operator overloading
- Method invocation
- Context manager support

https://docs.python.org/3/reference/datamodel.html#special-method-names

Slicing

Syntax

sequence*[start:stop[:step]]*

start
Optional. Starting index of the slice. Defaults to 0.
stop
Optional. The last index of the slice or the number of items to get. Defaults to *len(sequence)*.
step
Optional. Extended slice syntax. Step value of the slice. Defaults to 1.

```
def __getitem__():

def __len__():
```

A sequence is an object that has `__getitem__()` and `__len__()`

https://python-reference.readthedocs.io/en/latest/docs/brackets/slicing.html

Examples

```
>>> "ABCD"[0:2]   # 'AB'
>>> "ABCD"[0:4:2] # 'AC'
>>> "ABCD"[1:]    # 'BCD'
>>> "ABCD"[:3]    # 'ABC'
>>> "ABCD"[1:3]   # 'BC'
>>> "ABCD"[::2]   # 'AC'
>>> "ABCD"[::-1]  # 'DCBA'
```

List Comprehension

Syntax

```
new_list = [expression for member in iterable]
```

evaluates to something

list, set, sequence, generator...

object or value in the iterable

Similar to:

```
new_list = []
for member in iterable:
    new_list.append(expression)
```

https://realpython.com/list-comprehension-python/

But wait! There's more syntax???

Only appends when `condition` is true

```
new_list = [expression for member in iterable (if condition)]
```

```
below_n = [i for i in data if i < n]
```

For more complicated logic, we can provide an `else` by using:

```
new_list = [expression (if-else) for member in iterable]
```

```
square_positives = [i*i if i > 0 else i for i in data]
```


https://realpython.com/list-comprehension-python/

Pros and cons

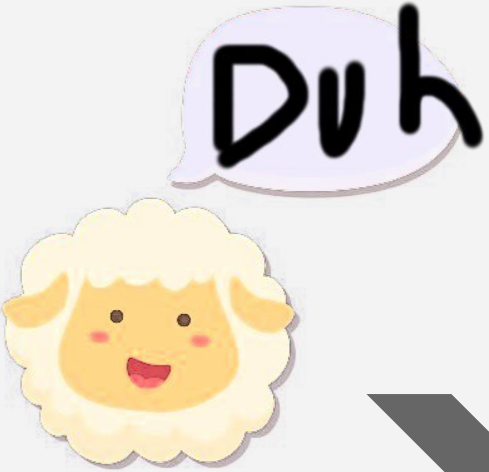
Pros

- Easier to read than nested loops
- Can replace maps and filters
- Focus on content, not setting up list

Cons

- Can get too complex
- Can't access earlier values
- Loads entire list into memory
  - Alternative: **Generator** comprehension

https://wiki.python.org/moin/Generators



Lambda Functions

What is a lambda function?

- Alternate type of function that isn't named (anonymous)
- Declared with syntax below

```python
lambda a, b: a + b  # sums two numbers
```

keyword

implicitly returned

input parameters

- Cannot contain statements, only an expression to return
- Per PEP8: Don't bind a lambda to an identifier (variable)

```python
add_one = lambda x: x + 1  # Does not follow PEP8
```

Why use lambda functions?

- Commonly used in higher-order functions
- Sometimes as predicates in functional programming

However, be careful not to over-use lambda functions

There’s often a better choice

Beautiful is better than ugly

DON'T

```
halve_evens_only = lambda nums: map(lambda i: i/2, filter(lambda i: not i%2, nums))
```

DO

```
def halve_evens_only(nums):
    return [i/2 for i in nums if not i % 2]
```

Virtual Environments (venvs)

Why use virtual environments?

- Cleaner development when working on multiple projects
- Separately install packages per project
    - No worries about conflicts between environments
- Support multiple different versions of Python

Common environment managers

- venv – included in Python >= 3.3
- conda – 3rd-party
- virtualenv – 3rd-party package
- pipenv – 3rd-party package

Key differences between managers

Global env directory:

- conda
- pipenv

```
~/
|-- .conda/
|    `-- envs/
|         |-- re2nfa/
|         `-- bunnyescape/
`-- projects/
     |-- re2nfa/
     `-- google/
          `-- bunnyescape
```

Local env directory:

- venv
- virtualenv

```
~/
`-- projects/
     |-- re2nfa/
     |    `-- .venv
     `-- google/
          `-- bunnyescape
               `-- .venv
```

Using venv

- Create a new environment

  ```
  > python -m venv .venv
  ```
- Create a new environment with version specification

  ```
  > python3.7 -m venv .venv
  ```
- Activate environment

  ```
  > source .venv/bin/activate
  (Windows)> .venv\Scripts\activate.bat
  ```
- List packages in environment

  ```
  > pip list
  ```
- Delete environment

  ```
  > rm -rf .venv
  (Windows)> rmdir /s .venv
  ```

Using conda

- Create a new environment

  ```
  > conda create python --name env_name
  ```
- Create a new environment with version specification

  ```
  > conda create python=3.7 --name env_name
  ```
- List environments

  ```
  > conda list env
  ```
- Activate environment

  ```
  > conda activate env_name
  ```
- List packages in environment

  ```
  > conda list
  ```
- Delete environment

  ```
  > conda env remove --name env_name
  ```

Unit Testing

Basic example of `unittest`

```
import unittest


class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())


if __name__ == '__main__':
    unittest.main()
```

Test cases inherit from TestCase base class

Use base class methods to test program correctness

(based on JUnit, hence the camelCase)

Test runner will discover and run test cases

Test Fixtures

```
import unittest


class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Executes before and after each test method

Custom error for test failure

Thanks!

Any questions?

WARNING

Bonus topics beyond this point

Bonus:
String Formatting

TLDR: Use f-strings

```
f'like this: {var} or {func(param)} or {a * b}'
```

Ways to format strings

- `printf` style:

```
some_string = 'Never gonna give %s up' % name
print('Iteration %d: value = %.10f' % (i, val))
```

- `str.format` style:

```
some_string = 'Never gonna give {} up'.format(name)
print('Iteration {}: value = {:.10f}'.format(i, val))
```

- f-string style:

```
some_string = f'Never gonna give {name} up'
print(f'Iteration {i}: value = {val:.10f}')
```

Bonus:
Custom Exceptions

Writing Custom Exceptions

```python
# exceptions.py

class ShoeError(Exception):
    """ base custom shoe exception """

class UntiedShoelaceError(ShoeError):
    """ You could fall """

class WrongFootError(ShoeError):
    """ When you try to wear your left shoe on your right foot """



# Raising that exception somewhere else
raise UntiedShoelaceError
```

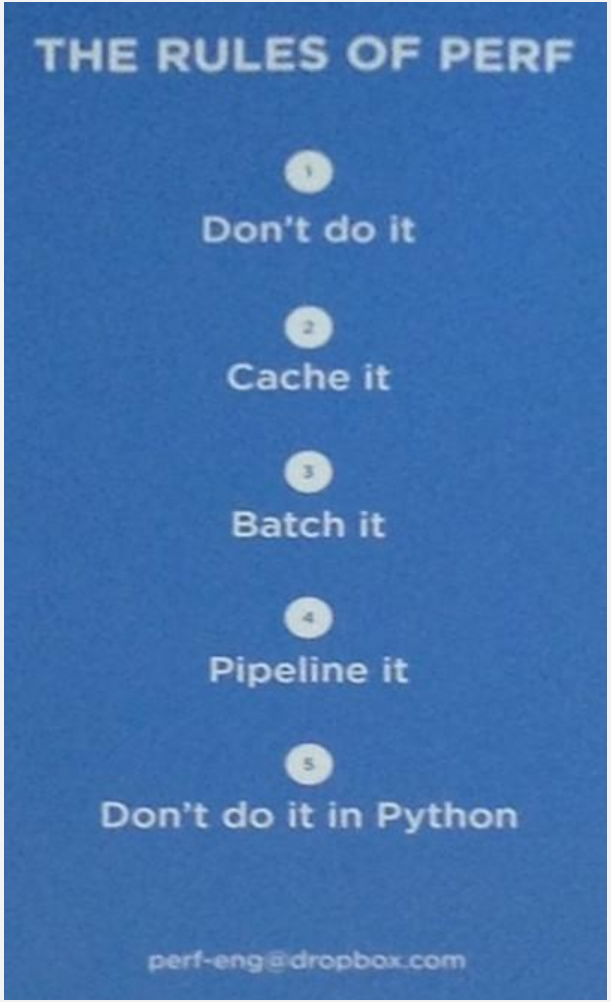
Using Custom Exceptions

```python
# elsewhere.py
from exceptions import ShoeError
from exceptions import UntiedShoelaceError
from exceptions import WrongFootError

try:
    wear_shoe()
except UntiedShoelaceError:
    print("Your laces are untied!")
except WrongFootError:
    print("Wrong foot dummy!")
except ShoeError:
    print("Shoes working wrong!")
except Exception as e:
    print(f"Some exception: {e}")
```

```python
# elsewhere_alt.py
import exceptions as exc

try:
    wear_shoe()
except exc.UntiedShoelaceError:
    print("Your laces are untied!")
except exc.WrongFootError:
    print("Wrong foot dummy!")
except exc.ShoeError:
    print("Shoes working wrong!")
except Exception as e:
    print(f"Some exception: {e}")
```

Bonus: Extending with C/C++

TLDR: Python go slow, C go fast

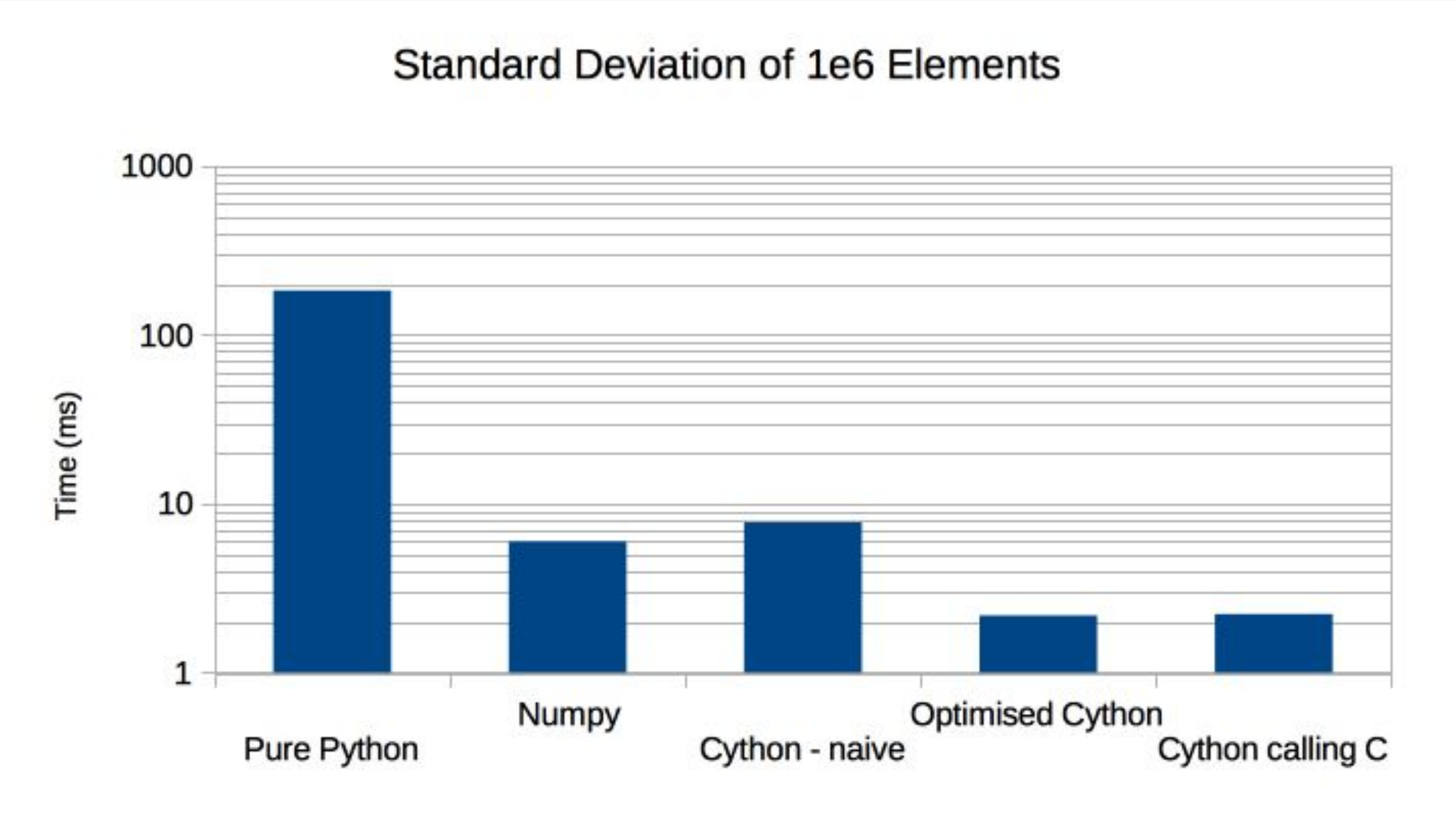


Why?

Can use C or C++ to extend Python. Common use cases:

- Increase performance
- Wrap existing C/C++ interfaces to be used more “Pythonic”-ly
- Low-level system access

https://notes-on-cython.readthedocs.io/en/latest/std_dev.html



Performance Comparison

https://notes-on-cython.readthedocs.io/en/latest/std_dev.html

How?

- Python API
- Cython (different from CPython!)
- pybind

Bonus:
Type Hinting

Statically Typed Python... Almost

Type hinting

- Makes it easier for linters to catch errors
- Doesn't actually enforce any type requirements

Syntax:

```
def repeat_string(s: str, repeats: int) -> str:
    return s * repeats

# hinting not really needed here
my_name: str = 'Garrett Walter'
```

Class Example

```python
class Vector:
    vectors_constructed = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        Vector.vectors_constructed += 1

    def get_tuple(self):
        return (self.x, self.y)

    def __add__(p):
        return Vector(self.x + p.x, self.y + p.y)
```

Class Example (with types!)

```python
from typing import Tuple

class Vector:
    vectors_constructed: int = 0

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y
        Vector.vectors_constructed += 1

    def get_tuple(self) -> Tuple[float, float]:
        return (self.x, self.y)

    def __add__(p: Vector) -> Vector:  # from version >=3.7
        return Vector(self.x + p.x, self.y + p.y)
```

Type Aliases

Eww ugly code

```python
from typing import List, Tuple

def deal_hands(deck: List[Tuple[str, str]]) -> Tuple[
    List[Tuple[str, str]],
    List[Tuple[str, str]],
    List[Tuple[str, str]],
    List[Tuple[str, str]],
    ]:

    # Deal the cards in the deck into four hands
    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
```

Type Aliases

You can for instance create Card and Deck type aliases:

```
from typing import List, Tuple

Card = Tuple[str, str]
Deck = List[Card]
```

Using these aliases, the annotations of deal_hands() become much more readable:

```
def deal_hands(deck: Deck) -> Tuple[Deck, Deck, Deck, Deck]:
    # Deal the cards in the deck into four hands
    return (deck[0::4], deck[1::4], deck[2::4], deck[3::4])
```

Bonus: Concurrency

The GIL (Global Interpreter Lock)

- In essence, just a Mutex
- Threads need to lock the mutex to execute py-bytecode
- Blocks CPU bound operations
- Avoids race conditions among multiple threads

Threading

- Can be run on multiple cores of a CPU
- Runs under the GIL
- All threads share the same memory
- i.e. able to use the same objects directly
- Does not block on I/O bound operations

Threading Example

```python
import concurrent.futures as cf
import threading
import requests

threadLocal = threading.local()

def get_session():
    if not hasattr(threadLocal, "session"):
        threadLocal.session = requests.Session()
    return threadLocal.session

def do_web(thing):
    session = get_session()
    pass # do something with session

def do_concurrent(things):
    with cf.ThreadPoolExecutor(max_workers=5) as executor:
        executor.map(do_web, things)

do_concurrent([1,2,3])
```

Multiprocessing

- Alternative to Threading
- Multiple Threads => Multiple Interpreters
- Procs can run on multiple cores
- No shared memory between any procs
- Communication becomes expensive
- Shares the same synchronization primitives as Threading

Multiprocessing Example

```python
import requests
import multiprocessing

session = None

def set_global_session():
    global session
    if not session:
        session = requests.Session()

def do_web(thing):
    pass # do something with session (global)

def do_concurrent(things):
    with multiprocessing.Pool(
        initializer=set_global_session
    ) as pool:
        pool.map(do_web, things)

do_concurrent([1,2,3])
```

Asyncio

- Method of passing CPU execution rights
- Never blocks
- Runs on one thread
- I.e. useful for single-core hardware
- 2 Task list and Event Loop

Asyncio Example

```python
import asyncio
import aiohttp

async def do_web(session, thing):
    pass # do something with session

async def do_concurrent(things):
    async with aiohttp.ClientSession() as session:
        tasks = []
        for thing in things:
            task = asyncio.ensure_future(
                do_web(session, thing)
            )
            tasks.append(task)
        await asyncio.gather(*tasks)

asyncio.get_event_loop().run_until_complete(
    do_concurrent([1,2,3])
)
```

Use Cases

- Both Threading and Asyncio don’t block on sockets
- Threading uses multiple cpus for concurrency, Asyncio uses one
- Multiprocessing is much faster for CPU bound operations
- Asyncio has an arguably nicer programming interface

In essence:

- CPU Bound => Multiprocessing
- I/O Bound, Fast I/O, Limited Number of Connections => Threading
- I/O Bound, Slow I/O, Many Connections => Asyncio

Bonus:
Iterators and Generators

Iterators

An ***iterator*** is an object that iterates over the sequence of an *iterable*

Implement Python's ***iterator*** protocol using `iter()` and `next()`

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through the elements using next()
print(next(my_iter))       # 4

#can iterate with for loop
for i in my_iter:
    print(i)               # 7 0 3
```

Iterable types:

- Lists
- Dictionaries
- Strings
- Tuples
- Files

Generators

Powerful iterators

Generator functions requires the use of the keyword *yield*

```python
def numberGenerator(n):
    number = 0
    while number < n:
        yield number
        number += 1

# create an generator object
myGenerator = numberGenerator(3)

# next() will iterate through
print(next(myGenerator))      # 0
print(next(myGenerator))      # 1
```

We can iterate using for or while loops:

```python
# both loops print same output
for i in numberGenerator(10):
    print(i)

while counter < 10:
    print(next(myGenerator))
    counter += 1
```

Generators vs Iterators

When to use what?

Though iterators and generators are set up differently, they both iterate over iterables and both seemingly produce the same results

Use **Generators** when:

- You do NOT have the data in memory AND there is a lot to iterate over
- You do not know what is going to come next

Use **Iterators** when:

- You already have the data in memory

Thanks again!

Any questions?

Other places to learn

Official documentation: https://docs.python.org/3/

Official wiki: https://wiki.python.org/

Python Crash Course: https://ehmatthes.github.io/pcc_2e/

LearnXinYminutes: https://learnxinyminutes.com/docs/python3/

Automate the Boring Stuff: https://automatetheboringstuff.com/2e/

Libraries to look into

Creating games:	`pyglet, pygame`
Creating GUIs:	`TkInter, PyQt, wxPython`
Data analysis:	`pandas, NumPy`
Data visualization:	`Matplotlib, Bokeh`
Taking CLI arguments:	`argparse, sys.argv`
Web scraping:	`requests, beautifulsoup4, scrapy`
Packaging as executable:	`PyInstaller`
Machine learning:	`TensorFlow, PyTorch, OpenAI, scikit-learn`
Web frameworks:	`Django, Flask, Pyramid`

Sources

- https://smallguysit.com/index.php/2017/10/28/python-benefits-using-virtual-environment/
- https://github.com/conda/conda/blob/master/docs/source/user-guide/conda-cheatsheet.pdf
- https://www.python.org/dev/peps/
- https://aaronlelevier.github.io/virtualenv-cheatsheet/
- https://docs.python-guide.org/dev/virtualenvs/
- https://docs.python.org/3.8/library/multiprocessing.html
- https://docs.python.org/3.8/library/threading.html
- https://docs.python.org/3.8/library/asyncio.html
- https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/
- https://realpython.com/python-concurrency/
- https://gist.github.com/evandrix/2030615
- https://docs.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2019
- https://docs.python.org/3/extending/index.html
- And more…