Real-World Python by Example

A no-nonsense and example-based Python course for beginners and intermediates alike

Nick Russo

Network and Automation Engineer

http://njrusmc.net

Pearson

About This Class

- Lots of
  - Pragmatic examples
  - Hands-on demonstrations
- Absolutely no
  - Academic "foo" and "bar" examples
  - Gimmicks and trivia
- Have questions? Please:
  - Use Q&A widget, not attendee/group chat
  - Keep them timely and relevant
  - Don't ask about your specific job

Section 1: Python Introduction

Data types

Operators

Packages and modules

Terminal input/output

Basic Data Types

- >>> my_var = 5 # an integer (int)
- >>> my_var = 5.3 # a floating point (float)
- >>> my_var = 'word' # a string (str)
- >>> my_var = True # a boolean (bool)

How to tell?

- >>> type(5) <class 'int'>
- >>> type(5.3) <class 'float'>
- >>> type('word') <class 'str'>
- >>> is_raining = False
- >>> type(is_raining) <class 'bool'>

How to test?

- >>> isinstance(5, int) True
- >>> isinstance('5', int) False
- >>> isinstance('5', str) True
- >>> my_gpa = 3.51
- >>> isinstance(my_gpa, int) False
- >>> isinstance(my_gpa, float) True

Why should I care?

- >>> a_string = '1'
- >>> an_int = 2
- >>> a_string + an_int

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Converting between types

- Known as "type conversion"

- >>> 5 + 5.4 # implicit int to float

10.4

- >>> int('1') + 2 # explicit str to int

3

Basic Arithmetic

- >>> 5 + 5 10 # addition
- >>> 5 - 8 -3 # subtraction
- >>> 5 * 6 30 # multiplication
- >>> 5 / 2 2.5 # division w/ decimal

Basic Arithmetic

- >>> 5 // 2 2 # division w/o decimal
- >>> 5 % 2 1 # modulus/remainder
- >>> 5 ** 3 125 # exponentiation

Order of Operations

- Standard PEMDAS
- >>> 5 * 2 ** 2 + 1 21
- >>> 5 * 2 ** (2 + 1) 40
- >>> (5 * 2) ** 2 + 1 101

Comparing Values

- >>> 5 == 5 True # test for equality
- >>> 5 != 5 False # test for inequality
- >>> 5 < 5 False # less than
- >>> 5 <= 5 True # less than or equal to
- >>> 5 > 5 False # greater than
- >>> 5 >= 5 True # greater than or equal to

Assignment Techniques

- >>> money = 5 # sets variable money to 5
- >>> money += 3 # increment money by 3
- >>> money -= 2 # decrement money by 2
- Others such as *=, /=, and **= do what you'd expect

Boolean Logic

- >>> 5 == 5 and 6 > 7 True and False: False
- >>> 5 == 5 or 6 > 7 True or False: True
- >>> not True False
- >>> 5 in [1, 3, 5, 7] True
- # membership or containment

Common Collections

- >>> my_var = {'a', 'b'} # a set (no order, no dup)
- >>> my_var = ['a', 'b', 'b'] # a list (ordered, can dup)
- >>> my_var = (1, 2) # a tuple (immutable list)
- >>> my_var = {'key': 'value'} # a dictionary (kv pairs)

Accessing Elements from a list or tuple

- Always starts at 0 and ends at length-1

- >>> languages = [

```
    'python',          # languages[0]
    'java',            # languages[1]
    'c',               # languages[2]
    'ruby'             # languages[3]
]
```

List Element Examples

- >>> languages = ['python', 'java', 'c', 'ruby']

- >>> languages[0]

'python'

- >>> languages[3]

'ruby'

Bad Indexing

- >>> languages = ['python', 'java', 'c', 'ruby']

- >>> languages[4]

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Accessing Dictionary Values

- >>> animals = {'cat': 'a feline', 'dog': 'a canine'}

- >>> animals['cat']

'a feline'

- >>> animals['dog']

'a canine'

Absent Key Attempt

- >>> animals = {'cat': 'a feline', 'dog': 'a canine'}

- >>> animals['goat']

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'goat'
```

Type Mixing

- All of these are technically valid
- >>> my_col = (1, 3, 'dog')
- >>> my_col = [1, 3, 'dog', 'cat', 2.3, False]
- >>> my_col = {'name': 'joe', 1: 2, 'cars': ['ford', 'bmw']}

Modules

- A single file
- Code that can be imported from elsewhere
- Module name is same as filename, minus .py

Importing Common Modules

- >>> import sys
- >>> type(sys)

<class 'module'>

- >>> print(sys.version)

3.10.5 (v3.10.5:f377153967, Jun 6 2022, 12:36:10) ...

- >>> sys.platform

'darwin'

Module Usage with "from"

- >>> import math
- >>> math.pi

3.141592653589793

- >>> from math import pi
- >>> pi

3.141592653589793

Working with Packages

- Package: hierarchy of modules

- $ tree shapes

```
shapes
|-- circle.py
|-- rectangle.py
`-- shape.py
```

Importing with Packages

- Still uses import
- General idea: "from package.module import item"
- >>> from shapes.rectangle import Rectangle
- >>> from shapes.circle import Circle

Basic print() Usage

- >>> print('I am happy')

I am happy

- >>> mood = 'happy'
- >>> print('I am ' + mood) # concatenation, not addition

I am happy

Using str.format() and f-strings

- >>> mood = 'happy'

- >>> print('I am {0}'.format(mood))

I am happy

- >>> print(f'I am {mood}')

I am happy

Collecting Interactive Input

- >>> name = input('enter your name: ')

enter your name: Nick

- >>> age = input('enter your age: ')

enter your age: 36

- >>> print(f'hi, I am {name} and {age} years old')

hi, I am Nick and 36 years old

Processing Command Line Arguments

- $ cat cli.py

```
import sys
print(sys.argv)
total = int(sys.argv[1]) + int(sys.argv[2])
print(f'sum is {total}')
```

- $ python cli.py 2 4

```
['cli.py', '2', '4']
sum is 6
```

Section 2: Fundamental Techniques

Conditionals

Iteration

Slicing

Functions

Error handling

Conditional Logic using "if"

- $ cat cond1.py

```
cars = ['bmw', 'ford', 'gmc']
if 'bmw' in cars:
    print('My first choice is BMW')      # indent 4 spaces
print('All done!')
```

Conditional Logic using "if"

- $ python cond1.py

```
My first choice is BMW        # because 'if' was true
All done!                     # always runs
```

Additional Clauses using "elif"

- $ cat cond2.py

```
cars = ['kia', 'ford', 'gmc']
if 'bmw' in cars:
    print('My first choice is BMW')
elif 'ford' in cars:
    print('My second choice is Ford')
print('All done!')
```

Additional Clauses using "elif"

- $ python cond2.py

```
My second choice is Ford        # because 'elif' was true
All done!                       # always runs
```

Default Clause with "else"

- $ cat cond3.py

```
cars = ['kia', 'nissan', 'gmc']
if 'bmw' in cars:
    print('My first choice is BMW')
elif 'ford' in cars:
    print('My second choice is Ford')
else:
    print('No cars I like')
print('All done!')
```

Default Clause with "else"

- $ python cond3.py

```
No cars I like          # because 'else' was true
All done!               # always runs
```

Iteration with "for"

- Good when number of iterations is/can be known

- $ cat for1.py

```
cars = ['bmw', 'ford', 'gmc']
for car in cars:
    print(f'I have a {car}')
```

Iteration with "for"

- $ python for1.py

I have a bmw

I have a ford

I have a gmc

Simple Linear Search

- $ cat for2.py

```
cars = ['bmw', 'ford', 'gmc']
for car in cars:
    if car == 'ford':
        print('Found ford')
```

- $ python for2.py

```
Found ford
```

Stopping Early

- $ cat for3.py

```
cars = ['ford', 'gmc', 'bmw', 'fiat', 'kia', 'nissan', 'lincoln']
for car in cars:
    print('Loop')
    if car == 'ford':
        print('Found ford')
        break
```

Stopping Early

- Loop stopped executing once found

- $ python for3.py

Loop

Found ford

Parallel Iteration with zip()

- Think of a "list of tuples"
- pair1, pair2, pair3, etc.

- $ cat zip.py

```
sizes = ['small', 'medium', 'large']
animals = ['frog', 'wolf', 'elephant']
for size, animal in zip(sizes, animals):
    print(f'I see a {size} {animal}')
```

Parallel Iteration with zip()

- $ python zip.py

I see a small frog

I see a medium wolf

I see a large elephant

Parallel Iteration with dict.items()

- cat items.py

```
animals = {
    'small': 'frog',
    'medium': 'wolf',
    'large': 'elephant'
}
for size, animal in animals.items():
    print(f'I see a {size} {animal}')
```

Parallel Iteration with dict.items()

- $ python items.py

I see a small frog

I see a medium wolf

I see a large elephant

Iteration with "while"

- Good when number of iterations is not/cannot be known

- $ cat while.py

```
num = 0
while int(num) % 2 == 0:
    num = input('enter odd number: ')
print('good job!')
```

Iteration with "while"

- $ python while.py

enter odd number: 2

enter odd number: 4

enter odd number: 5

good job!

Slicing (AKA substring)

- >>> ip_addr = '203.0.113.68/24'

- >>> ip_addr[0]

'2'

- >>> ip_addr[3]

'.'

- >>> ip_addr[-1] # count backwards 1 char

'4'

Slicing More than One Character

- >>> ip_addr = '203.0.113.68/24'

- >>> ip_addr[:12] # grab 0 (inclusive) to 12

'203.0.113.68'

- >>> ip_addr[12:] # grab 12 (inclusive) to end

'/24'

- >>> ip_addr[6:9] # grab 6 (inclusive) to 9

'113'

Slicing More than One Character

- >>> ip_addr = '203.0.113.68/24'

- >>> ip_addr[:-3] # from start, go until 3rd to end

'203.0.113.68'

- >>> ip_addr [-3:] # from 3rd to end, go to end

'/24'

- >>> ip_addr [-9:-6] # from 9th to end until 6th to end

'113'

Slicing with Lists

- >>> cars = ['bmw', 'ford', 'gmc', 'nissan', 'kia']

- >>> cars[2:4] # grab 2 (inclusive) to 4

['gmc', 'nissan']

- >>> cars[-4] # 4th from right until the end

'ford'

- >>> cars[:3] # grab first 3 items

['bmw', 'ford', 'gmc']

Defining Custom Functions

```
def square_plus1(x):
    return x ** 2 + 1

print(f'2^2 +1 is {square_plus1(2)}')
print(f'3^2 +1 is {square_plus1(3)}')
```

- $ python func.py

```
2^2 + 1 is 5
3^2 + 1 is 10
```

Measuring Length with len()

- >>> len('python') 6
- >>> len([1, 3, 5, 7, 9]) 5
- >>> len({'k1': 'v1', 'k2': 'v2'}) 2

- >>> len(1)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Function Chaining

- >>> course = ' python example '

- >>> course.upper()

' PYTHON EXAMPLE '

- >>> course.upper().strip()

'PYTHON EXAMPLE'

- >>> course.upper().strip().startswith('P')

True

Error Handling - Stack

- Concept of function call "stack"
- First in, last out
- Newest (topmost) entries shown last

Error Handling - Stack

- $ cat stack1.py

```
import sys
def get_endian():
    return sys.byteorder

def print_result():
    print (get_endian())

print_result()
```

Error Handling – Stack execution

- Call sequence
  - top-level called print_result()
  - print_result() called get_endian()
  - get_endian() returned the endian-ness of the system

- python stack1.py

little

Error Handling – Bad Index

- $ cat stack2.py

```
import sys
def get_endian():
    return sys.byteorder[42518]

def print_result():
    print (get_endian())

print_result()
```

Behold, the Traceback (AKA Stack Trace)

```
$ python stack2.py
Traceback (most recent call last):
  File "stack2.py", line 8, in <module>
    print_result()
  File "stack2.py", line 6, in print_result
    print (get_endian())
  File "stack2.py", line 3, in get_endian
    return sys.byteorder[42518]
IndexError: string index out of range
```

Error Handling – Try/Except

```
$ cat stack3.py

# get_endian() snipped for brevity
def print_result():
    try:
        print (get_endian())        # error is raised here
    except IndexError as exc:
        print(exc)
print_result()
```

Error Handling – Try/Except execution

- Exception is "handled"; allow the program to continue

- $ python stack3.py

string index out of range

Raising Errors Yourself

- $ cat stack4.py

```
import sys
def get_endian():
    # return sys.byteorder[42518]
    raise NotImplementedError("do not use!")

def print_result():
    print (get_endian())
print_result()
```

Raising Errors Yourself

- $ python stack4.py

```
Traceback (most recent call last):
  File "stack4.py", line 9, in <module>
    print_result()
  File "stack4.py", line 7, in print_result
    print (get_endian())
  File "stack4.py", line 4, in get_endian
    raise NotImplementedError("do not use!")
NotImplementedError: do not use!
```

Section 3: Fundamental Solution Review

Assemble existing knowledge into a project

Section 4: Optimizations and Troubleshooting

List comprehensions

Type attributes

Troubleshooting techniques

4 Lines of Code for a Simple Task?

- $ cat sqfor.py

```
my_squares = []
for i in range(5):
    my_squares.append(i ** 2)
print(my_squares)
```

- $ python sqfor.py

```
[0, 1, 4, 9, 16]
```

1 Line of Code with List Comprehensions

- Remember the magic word: CHOOSE or SELECT

- $ cat sqlc.py

```
print([i ** 2 for i in range(5)])
```

- $ python sqlc.py

```
[0, 1, 4, 9, 16]
```

More List Comprehension Examples

- cars = ['ford', 'gmc', 'bmw', 'fiat', 'kia']

- >>> [car.upper() for car in cars]

['FORD', 'GMC', 'BMW', 'FIAT', 'KIA']

- >>> [car[:2] for car in cars]

['fo', 'gm', 'bm', 'fi', 'ki']

- >>> [car for car in cars if car.startswith('f')]

['ford', 'fiat']

Type Attributes using dir()

- >>> str.__dir__

<method '__dir__' of 'object' objects>

- >>> dir(str)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',

((snip))

'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Reviewing Documentation

- >>> print(str.__doc__)

str(object='') -> str

str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. ((snip))

A Custom Example

- >>> import fundamental
- >>> print(fundamental.get_units.__doc__)

Return the unit of measure, either centimeters (cm) or inches (in) based on user input via command line arguments or via interactive input collection.

A Custom Example

- >>> import fundamental
- >>> help(fundamental)

- "man page" style

```
NAME
    fundamental

DESCRIPTION
    Author: Nick Russo
    File: fundamental.py
    Purpose: Fundamental techniques demonstration.

FUNCTIONS
    get_units(argv)
        Return the unit of measure, either centimeters (cm) or
        inches (in) based on user input via command line arguments
        or via interactive input collection.

    main(argv)
        Execution starts here.

DATA
    pi = 3.141592653589793

FILE
    /Users/nicholasrusso/Desktop/Python/shapes/fundamental.py
```

Troubleshooting with print()

- No special tools or knowledge required
- Easily toggled using comments (#)
- Display value of questionable variables

What's the Problem?

- $ cat debug1.py

```
def factorial(n):
    total = 0
    for i in range(n):
        total *= i + 1
    return total

print(factorial(5))
print(factorial(8))
```

Doesn't Work ...

- $ python debug1.py

0

0

Using print() and f-strings to Help

- $ cat debug2.py

```
def factorial(n):
    total = 0
    for i in range(n):
        total *= i + 1
        # print(f"i={i}, total={total}")      # Python 3.6+
        print(f"{i=}, {total=}")              # Python 3.8+
    return total
```

Total is Stuck at 0 ... Why?

- $ python debug2.py

i=0, total=0

i=1, total=0

i=2, total=0

i=3, total=0

i=4, total=0

((snip))

Successive Multiplication Should Start at 1

- $ cat debug3.py

```
def factorial(n):
    total = 1                          # changed 0 to 1
    for i in range(n):
        # print(f"i={i}, total={total}")      # Python 3.6+
        print(f"{i=}, {total=}")               # Python 3.8+
        total *= i + 1
    return total
```

It Works!

- $ python debug3.py

i=0, total=1

i=1, total=2

i=2, total=6

i=3, total=24

i=4, total=120

120

((snip))

40320

Python Debugger (pdb)

- Set a trace (AKA breakpoint)
- Built-into Python; no IDE needed
- Four basic operations
  - list (l)
  - next (n)
  - step (s)
  - continue (c)

Python Debugger (pdb)

- $ cat pdeb1.py

```
def get_last_char(word):
    # Legacy technique, pre-Python 3.7
    # import pdb; pdb.set_trace()

    breakpoint()          # Python 3.7+
    return word[len(word)]

print(get_last_char('program'))
```

Python Debugger (pdb)

- $ python pdeb1.py

```
> pdeb1.py(6)get_last_char()
-> return word[len(word)]
(Pdb) next
IndexError: string index out of range
> pdeb1.py(6)get_last_char()
-> return word[len(word)]
(Pdb) print(len(word), word)
7 program
```

The Fix

- $ cat pdeb2.py

```
def get_last_char(word):
    # Legacy technique, pre-Python 3.7
    # import pdb; pdb.set_trace()

    breakpoint()          # Python 3.7+
    return word[-1]       # count from right with -1

print(get_last_char('program'))
```

Python Debugger (pdb)

- $ python pdeb2.py

```
> pdeb2.py(6)get_last_char()
-> return word[-1]
(Pdb) n
--Return--
> pdeb2.py(6)get_last_char()->'m'
-> return word[-1]
(Pdb) c
m
```

Step vs. Next

- $ cat pdeb3.py

```
def door(state):
    print(f'door is {state}')

# import pdb; pdb.set_trace()
breakpoint()
door('open')
```

Next is "Step Over"

- $ python pdeb3.py

```
> pdeb3.py(5)<module>()
-> door('open')
(Pdb) next
door is open
--Return--
> pdeb3.py(5)<module>()->None
```

Step is "Step Into"

- $ python pdeb3.py

```
> pdeb3.py(5)<module>()
-> door('open')
(Pdb) step
--Call--
> pdeb3.py(1)door()
-> def door(state):
```

Section 5: Object Oriented Programming (OOP)

Classes

Objects

Methods

Inheritance

Abstraction

Polymorphism

Fundamentals of OOP

- Represent objects as individual entities
- A class defines a blueprint for an object
- An object is an instance of a class
- The constructor initializes an object's data

Class Example

- $ cat dog.py

```
class Dog:
    def __init__(self, color, size):
        self.color = color
        self.size = size
```

Instantiating an Object

- $ cat dogtest.py

```
from dog import Dog
my_dog = Dog('brown', 'large')
print(type(my_dog), my_dog.color, my_dog.size)
```

- $ python dogtest.py

```
<class 'dog.Dog'> brown large
```

Adding Methods

- $ cat dog.py

```
class Dog:
    def __init__(self, color, size):
        self.color = color
        self.size = size
    def get_info(self):
        return f'dog is {self.size} and {self.color}'
```

Invoking Methods

- $ cat dogtest.py

```
from dog import Dog
my_dog = Dog('brown', 'large')
print(my_dog.get_info())
```

- $ python dogtest.py

```
dog is large and brown
```

List of Dog Objects

- $ cat dogtest.py

```
from dog import Dog
my_dogs = [
    Dog('black', 'large'),    # Lucy
    Dog('white', 'small')    # Lily
]
for my_dog in my_dogs:
    print(my_dog.get_info())
```

List of Dog Objects

- $ python dogtest.py

dog is large and black

dog is small and white

Inheritance – Base Class

- Classes can be hierarchical
  - Think animals, cars, shapes, governments, networks, etc.

- $ cat animal.py

```
class Animal:
    def __init__(self, size):
        self.size = size.upper()
```

Inheritance – Child Class

- $ cat dog.py

```
from animal import Animal
class Dog(Animal):
    def __init__(self, bark_type, size):
        self.bark_type = bark_type
        super().__init__(size)
    def get_info(self):
        return f'dog is {self.size} with a {self.bark_type} bark'
```

Inheritance – Child Object

- $ cat dogtest.py

```
from dog import Dog
my_dog = Dog('loud', 'small')
print(my_dog.get_info())
```

- $ python dogtest1.py

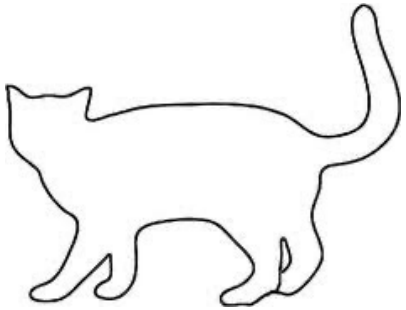
```
dog is SMALL with a loud bark
```

Abstraction

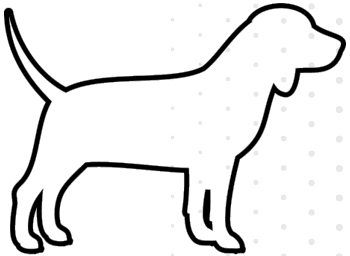
- Not all classes should be instantiated
- What does a generic animal look like?
  - ... or a generic shape?
  - ... or a generic government?
  - ... or a generic vehicle?

The Sniff Test: Try to Visualize It

?



```
class Animal
    __init__()
```



```
import animal
class Cat
    __init__()
    get_purr()
```

```
import animal
class Dog
    __init__()
    get_bark()
```

Abstract Class Example

- $ cat animal.py

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod                    # decorator
    def __init__(self, size):
        self.size = size.upper()
```

Instantiating an Abstract Class ...

- Should this work?

- $ cat animaltest1.py

```
from animal import Animal
my_animal = Animal('small')
print(my_animal.size)
```

Does Not Make Sense!

- $ python animaltest1.py

```
Traceback (most recent call last):
File "animaltest1.py", line 2, in <module>
    my_animal = Animal('small')
TypeError: Can't instantiate abstract class Animal with abstract methods __init__
```

Still Use the Dog Object the Same Way!

- $ cat dogtest.py

```
from dog import Dog
my_dog = Dog('loud', 'small')
print(my_dog.get_info())
```

- $ python dogtest.py

```
dog is SMALL with a loud bark
```

Polymorphism

- Classes with a common base class treated similarly
- Invoke base class abstract methods
- Different results based on child implementation

New Base Class get_info() Method

- $ cat animal.py

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def __init__(self, size):
        self.size = size.upper()
    @abstractmethod
    def get_info(self):            # children must implement it
        pass
```

New Cat Class

- $ cat cat.py

```
from animal import Animal
class Cat(Animal):
    def __init__(self, purr_type, size):
        self.purr_type = purr_type
        super().__init__(size)
    def get_info(self):
        return f'cat is {self.size} with a {self.purr_type} purr'
```

Create Generic Animal List

- $ cat animaltest2.py

```
import dog, cat
my_animals = [
    dog.Dog('friendly', 'big'),
    cat.Cat('meow', 'tiny')
]
for my_animal in my_animals:
    print(my_animal.get_info())
```

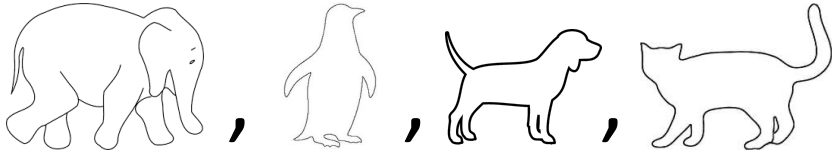
Polymorphism in Action

- $ python animaltest2.py

dog is BIG with a friendly bark

cat is TINY with a meow purr

Polymorphism Visualized

>>> animals = []

```
class Animal
    def __init__(...)
@abstractmethod
    def get_info()
```

each animal implements
get_info() differently

Section 6: File Input/Output (I/O)

Plain text formatting

YAML formatting

JSON formatting

Reading/writing data

Opening/closing streams

Reading Data from a Plain Text File

```
$ cat mantextr.py
try:
    handle = open("office.txt", "r")
    data = handle.read()
    print(data)
except IOError as exc:
    print(exc)
finally:                    # always runs
    handle.close()
```

Sample File and Execution

- $ cat office.txt

My office has a bed, a desk, and 4 walls that are light blue in color.

- $ python mantextr.py

My office has a bed, a desk, and 4 walls that are light blue in color.

Writing Data to a Plain Text File

- $ cat mantextw.py

```
try:
    handle = open("house.txt", "w")
    handle.write("I live in a 3BD/2BR house which is white and blue in color\n")
except IOError as exc:
    print(exc)
finally:
    handle.close()
```

Writing Data to a Plain Text File

- $ python mantextw.py && cat house.txt

I live in a 3BD/2BR house which is white and blue in color

- Tip: the Bash "&&" symbol can be used to chain commands, assuming the previous one succeeded

Yet Another Markup Language (YAML) Syntax

- $ cat office.yml

```
name: office            # YAML allows comments
furniture:              # value is list of strings
  - bed
  - desk
walls:                  # value is another dict (nested)
  number: 4
  color: light_blue
```

Using a Context Manager with YAML

- $ cat withyamlr.py

```
import yaml                    # pip install pyyaml (see README.md)
with open("office.yml", "r") as handle:
    try:
        data = yaml.safe_load(handle)
        print(data)
    except yaml.YAMLError as exc:
        print(exc)
```

Using a Context Manager with YAML

- Output looks like Python syntax

- $ python withyamlr.py

{'name': 'office', 'furniture': ['bed', 'desk'], 'walls': {'number': 4, 'color': 'light_blue'}}

Writing Data to a YAML File

- $ cat withyamlw.py

```
import yaml
data = {
    'bedrooms': 3,
    'bathrooms': 2,
    'color': ['blue', 'white']
}
with open('house.yml', 'w') as handle:
    yaml.dump(data, handle, default_flow_style = False)
```

Writing Data to a YAML File

- $ python withyamlw.py && cat house.yml

```
bathrooms: 2
bedrooms: 3
color:
  - blue
  - white
```

JavaScript Object Notation (JSON) Syntax

- $ cat office.json

```
{
  "name": "office",
  "furniture": [
    "bed",
    "desk"
  ],
  "walls": {
    "number": 4,
    "color": "light_blue"
  }
}
```

Using a Context Manager with JSON

```
$ cat withjsonr.py
import json
with open("office.json", "r") as handle:
    try:
        data = json.load(handle)
        print(data)
    except json.decoder.JSONDecodeError as exc:
        print(exc)
```

Using a Context Manager with JSON

- Identical output as seen with YAML!
- It's all internal Python data after the load() function

- $ python withjsonr.py

{'name': 'office', 'furniture': ['bed', 'desk'], 'walls': {'number': 4, 'color': 'light_blue'}}

Writing Data to a JSON File

- $ cat withjsonw.py

```
import json
data = {
    'bedrooms': 3,
    'bathrooms': 2,
    'color': ['blue', 'white']
}
with open('house.json', 'w') as handle:
    json.dump(data, handle, indent=4)
```

Writing Data to a JSON File

- $ python withjsonw.py && cat house.json

```
{
    "bedrooms": 3,
    "bathrooms": 2,
    "color": [
        "blue",
        "white"
    ]
}
```

Section 7: Complete Solution Review

Assemble existing knowledge into a project

Section 8: Unit Testing

Purpose of unit tests

unittest

pytest

Purpose of Unit Testing

- Catch problems at a basic level
- Check for regressions
- Required for DevOps (Continuous Integration)
- Code is easier to maintain

Section 9: Reviewing Unfamiliar Code

Apply your analytics skills and tools

Includes:

CSV files
Ternary operations
Splitting/joining of strings
Built-in data validation
CLI arguments and options

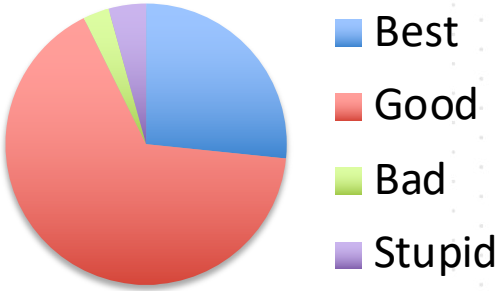
Secret Project Review

- Real-life scenario
  - Boss says "Hey, can you work on project XYZ?"
  - You always know some, but never all, of the tech involved

- How can you ....
  - Navigate the uncertainty?
  - Quickly/test validate your hypotheses?
  - Be productive without extensive training/research?

Opinion Polling

Votes (n=94)



Introducing Zhong 中

- Flashcard system to learn Mandarin Chinese

- Technical highlights
  - Reads Chinese (Hanzi), Pinyin, and English from CSV file
  - Prompts player to interpret Chinese, Pinyin, and narration
  - Prints correct answer in ...
    - Green if the player answered correctly
    - Red otherwise
  - Minor options to toggle Chinese, Pinyin, narration, etc.

Our Plan

- Selectively step through the code using ...
  - Interactive shell
  - pdb via breakpoint()
  - functions like dir(), help(), type(), and print()
  - Anything else that makes sense!

- https://github.com/nickrusso42518/zhong

Recap and Q&A

What we learned:

Many Python implementation techniques

Structured data reading and writing

Common OOP design patterns

Maintaining quality through testing

Logically evaluating an unknown program

Class Challenge

- Fork or clone the repository
  - https://github.com/nickrusso42518/slt-py-example

- Add a Triangle class to the "shapes" project
  - Notify me when complete
  - Twitter: @nickrusso42518
  - Email: njrusmc@gmail.com

- I'll grade your work with personalized feedback!