Python Next Steps: Functions and Scripting

Colby Witherup Wood
July 15 and 17, 2019

Goals for this workshop

Help you transition from interactive coding to ***scripts***

Give you tools to write code that is:

- Organized and Readable
- Reuseable
- Pythonic
- Ready to be shared - professionally, publicly, or casually
- Less stressful

Goals for this workshop

Secondary goals

- Learn what it means to be "pythonic"
- Hear terms that you will encounter as you improve
- Give some advice that I wish I had known when I was a beginning Python coder (this is only my own advice and you may get opposite advice from other coders, and as you improve)

What is interactive coding?

Interactive programming from wikipedia: "writing parts of a program while it is already active". We mean coding directly in a Python shell.

If you go into a Python IDE to work on a project or solve a problem and you start typing right in the Python shell… I'm going to try to break you of that habit.

Why should you cut back on interactive coding?

Code can't be edited, read, shared, reused, or searched.

Why should you cut back on interactive coding?

Code can't be edited, read, shared, reused, or searched.

Pieces of code aren't self-contained. This leads to unexpected errors like variables getting changed and files getting altered that cannot be unaltered. These are issues of ***scope***.

Why should you cut back on interactive coding?

Code can't be edited, read, shared, reused, or searched.

Pieces of code aren't self-contained. This leads to unexpected errors like variables getting changed and files getting altered that cannot be unaltered. These are issues of ***scope***.

Code can't be viewed at once, and you shouldn't have to dedicate brain power to remembering what you did (and why you did it) ten minutes ago (let alone 30 minutes ago, or yesterday, or three months ago when you last worked on this project).

What do you need to code?

What do you need to code?

A way to talk to your computer.
A command prompt, terminal, or bash shell.

What do you need to code?

A way to talk to your computer.
A command prompt, terminal, or bash shell.

That's it!

What do you need to code? What helps?

A way to talk to your computer. A command prompt, terminal, or bash shell.

What do you need to code? What helps?

A way to talk to your computer. A command prompt, terminal, or bash shell.

A shell that is specific to your programming language with some color coding and predictive properties, like iPython.

What do you need to code? What helps?

A way to talk to your computer. A command prompt, terminal, or bash shell.

A shell that is specific to your programming language with some color coding and predictive properties, like iPython.

A text editor with color coding and predictive properties specific to your coding language, for writing scripts.

What do you need to code? What helps?

A way to talk to your computer. A command prompt, terminal, or bash shell.

A text editor with color coding and predictive properties specific to your coding language, for writing scripts.

A shell that is specific to your programming language with some color coding and predictive properties, like iPython.

Tools for data visualization, debugging, and file management

What we will be using for this workshop

Command line

Python shell

Text Editor

When to use…

Command line

- **Moving around in directories, moving files and directories**
- **Running Python scripts**
- **Running Python scripts with arguments**
- **Creating pipelines of multiple Python scripts**
- **Creating pipelines that include both Python scripts and other software programs or scripts in other languages**
- **Submitting jobs on a remote server like Quest**

When to use…

Python shell

- **Testing short pieces of code to see if they work**
- **Troubleshooting pieces of code**
- **Using Python as a calculator**

When to use…

Text Editor

- **Writing code**
- **Reading code**
- **Running code IF you are using an IDE that allows you to run code from the text editor and IF no arguments are needed for the script. HOWEVER, if you are planning on sharing the script, it is best to run it from the command line**

What we will be using for this workshop:

Command line

Python shell

Text Editor

How to set up your workspace

IDE stands for Interactive Development Environment.

Jupyter Lab has a Terminal, Python shell, and Text editor.

Spyder and PyCharm have the Python shell and Text Editor. (You will need to open a separate command line shell.)

Or you can build your own workspace from a good text editor like BBEdit (Mac) or SciTE (Mac/Windows) and two command line windows - one running iPython or Python

Schedule for the workshop

1. Python etiquette Intro
2. Functions
3. Scripts
4. sys module
5. Error handling
6. Python etiquette 2

1. Python etiquette

1. Python etiquette

PEP: Python Enhancement Proposals

Some PEPs describe or propose new Python features.

Informational PEPs provide general guidance to the Python community. Users "are free to ignore Informational PEPs or follow their advice". Some Informational PEPs set guidelines for what is considered "pythonic" and create universal standards for professional Python coders (though each workplace may vary). **It's good to get in the habit of following these standards now.**

1. Python etiquette

In your Python shell, type:

```
import this
```

1. Python etiquette

PEP 20: The Zen of Python

Long time Pythoneer Tim Peters summarizes Python's guiding principles into 20 aphorisms, only 19 of which have been written down.

Aphorism - a pithy observation that contains a general truth

1. Python etiquette

PEP 8: Style Guide for Python Code

- indentation
- where to break up long lines of code
- where to put blank lines in scripts
- how to best import modules
- where to put spaces
- how to name variables

1. Python etiquette

Naming variables and functions

Use lowercase words separated by _

`animal_list` `animals` `name_address_dict`

`sort_animal_list()`

Avoid using l (lowercase L), O (uppercase o), and I (uppercase i) as single letter variables

1. Python etiquette

Naming variables and functions

For temporary variables you can use single letters

`i`   `B`

It is also good to stick to one system for naming files and folders. This style is recommended as it can be read on most computer systems and by most programs without problems.

`animalFile.txt`   `myScript.py`

2. Functions

2. Functions

Why use functions?

Organization and readability - break your code into modular chunks

Reusability - avoid repeating yourself

Scope - keep track of namespaces and avoid confusion with variable names

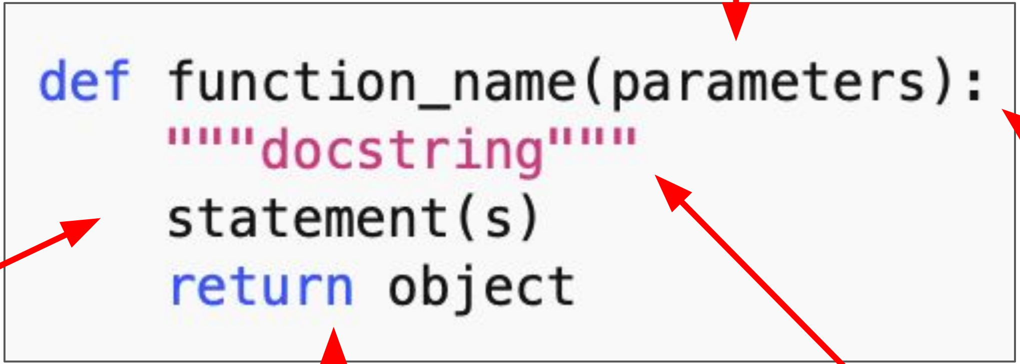
"Flat is better than nested"

2. Functions

Parts of a function

0, 1, or more parameters

the first line is a def statement



```
def function_name(parameters):
    """docstring"""
    statement(s)
    return object
```

colon

everything inside the function is indented

return statement is optional

a short description of your function

2. Functions

Write a function with no parameters.

Let's go to our Text Editor and write our first function. Open up a blank text document.

In some editors, to get the Python-specific color coding and predictive properties, you have to first save the document as a .py file. You can save this file as functionPractice.py

2. Functions

Write a function with no parameters.

```
def multiply_10_50():
    """print the product of 10 and 50"""
    product = 10 * 50
    print(product)
```

2. Functions

<u>Write a function with no parameters.</u>

To run your code, you might be able to run it straight from the Text Editor in some IDEs

OR copy the code and paste it in your Python shell

```
def multiply_10_50():

    """print the product of 10 and 50"""

    product = 10 * 50

    print(product)
```

2. Functions

Write a function with no parameters.

Why did nothing happen when you ran the code for your function?

2. Functions

Write a function with no parameters.

call your function in the Python shell:

```
multiply_10_50()
```

```
def multiply_10_50():
    """print the product of 10 and 50"""
    product = 10 * 50
    print(product)
```

2. Functions

Write a function with no parameters.

try:

```
print(product)
```

```
def multiply_10_50():
    """print the product of 10 and 50"""
    product = 10 * 50
    print(product)
```

2. Functions

Write a function with no parameters.

`product` is a variable that exists only inside your function. We call this the ***local*** scope. `product` does not exist in the ***global*** scope of your python shell because it has not been named in the ***global*** scope.

2. Functions

Write a function with no parameters.

If you want your local variable to be accessible in the global scope, you can do that, but you have to remember that every time you call your function, your variable could change in the global scope. Calling a variable globally has its uses, but be careful.

2. Functions

<u>Write a function with no parameters.</u>

you must state that you want the variable stored globally before you assign the variable

```
def multiply_10_50():
    """print the product of 10 and 50"""
    global product
    product = 10 * 50
    print(product)
```

2. Functions

<u>Write a function with one parameter.</u>

2. Functions

Write a function with one parameter.

```
def multiply_by_10(number):
    """print the product of 10 and any
    number"""
    product = 10 * number
    print(product)
```

2. Functions

Write a function with one parameter.

the variable included in the def statement must match exactly to the variable used in the body

```
def multiply_by_10(number):
    """print the product of 10 and any
    number"""
    product = 10 * number
    print(product)
```

2. Functions

Write a function with one parameter.

run the function, and then
call it with any number

```
multiply_by_10(100)

multiply_by_10(8)
```

```
def multiply_by_10(number):
    """print the product of 10 and any
    number"""
    product = 10 * number
    print(product)
```

2. Functions

<u>Write a function with multiple parameters.</u>

2. Functions

<u>Write a function with multiple parameters.</u>

```
def calculate_area(length, width):
    """print the area of a rectangle"""
    area = length * width
    print(area)
```

2. Functions

<u>Write a function with multiple parameters.</u>

run the function, and then call it with any two numbers

```
calculate_area(10, 20)

calculate_area(500, 3)
```

```
def calculate_area(length, width):

    """print the area of a rectangle"""

    area = length * width

    print(area)
```

2. Functions

Write a function that returns one object.

Usually you don't want to print your result to the screen, but you want to save it as a variable that can be used in the global space. We use `return` to tell our function which object or objects should be available to be named outside the function.

2. Functions

<u>Write a function that returns one object.</u>

reusing the last function, change the print function to a return statement, and change the docstring

```
def calculate_area(length, width):
    """return the area of a rectangle"""
    area = length * width
    return area
```

2. Functions

<u>Write a function that returns one object.</u>

run the function, and then call it with any two numbers, then try to print the area

```
calculate_area(10, 20)

print(area)
```

```
def calculate_area(length, width):

    """return the area of a rectangle"""

    area = length * width

    return area
```

2. Functions

Write a function that returns one object.

Assign the function to a variable. You can now use my_area like any variable.

```
def calculate_area(length, width):
    """return the area of a
    rectangle"""
    area = length * width
    return area
```

```
my_area = calculate_area(10, 20)
print(area)
my_area * 5
```

2. Functions

<u>Write a function that returns multiple objects.</u>

In the text editor, let's write a new function that returns two variables.

2. Functions

Write a function that returns multiple objects.

```
def calculate_area_acres(length, width):
    """return the area and acreage of a
    rectangle"""
    area = length * width
    acres = area / 43560
    return area, acres
```

2. Functions

<u>Write a function that returns multiple objects.</u>

run the function in the shell

```
def calculate_area_acres(length, width):
    """return the area and acreage of a
    rectangle"""
    area = length * width
    acres = area / 43560
    return area, acres
```

2. Functions

Write a function that returns multiple objects.

We can name multiple variables that are returned by a function like this:

```
def calculate_area_acres(length, width):

    """return the area and acreage

    of a rectangle"""

    area = length * width

    acres = area / 43560

    return area, acres
```

```
area1, acres1 = calculate_area_acres(10, 30)
```

The variables names you give will be assigned in the same order as they are given in the return statement of your function.

2. Functions

<u>You've learned how to write and call functions:</u>

- with no parameters
- with 1 parameter
- with more than 1 parameter
- that return 1 object
- that return more than 1 object

2. Functions

Parameter types

So far we've used numbers as our parameters and return objects. You can use any object as a parameter or return object, including strings, lists, and dictionaries.

Save the document we have been working in as functionPractice.py if you haven't saved it already.

We will continue in the same file.

2. Functions

Practice writing functions #1

Write a function that takes a list as the only parameter. The function should return the first item of the list.

After you have written your function, run it in the Python shell, and test it on a list. Here is a list you can use, but any will work.

```
tree_list = ["red maple", "red oak", "white
oak", "buckeye", "blue spruce", "magnolia"]
```

2. Functions

<u>Practice writing functions #2</u>

Write a function called `return_nth` that takes two parameters, a list and an integer. Return the item in the list that corresponds to the integer provided when calling the function.

Remember that Python starts counting with zero.

After you have written your function, run it in the Python shell, and test it on the same list you used in the previous example.

2. Functions

<u>Writing a function with unknown number of parameters.</u>

Sometimes you don't know how many parameters you'll be given, but you want to do the same thing to all of them.

`*args` can be used as the parameter in your `def` statement.

2. Functions

Writing a function with unknown number of parameters.

Let's say we want to write a function that can calculate the acreage of many rectangles. We want to give the function multiple tuples, each with a length and a width, like this:

```
(10, 100), (20, 60), (50, 80)
```

2. Functions

<u>Writing a function with unknown number of parameters.</u>

your function must then include a loop through the "args"

```
def calculate_acres_tuples(*args):
    """take an unknown number of tuples with
    length and width, and print acreage"""
    for arg in args:
        acres = (arg[0] * arg[1]) / 43560
        print(acres)
```

2. Functions

Writing a function with unknown number of parameters.

run the function, and call it with multiple tuples

```
def calculate_acres_tuples(*args):
    """take an unknown number of tuples with
    length and width, and print acreage"""
    for arg in args:
        acres = (arg[0] * arg[1]) / 43560
        print(acres)
```

```
calculate_acres_tuples((10, 100), (20, 60), (50, 80))
```

3. Scripts

3. Scripts

What is a script?

A Python script is a text file with the extension .py that contains 1 or more functions.

A script can also be called a program.

3. Scripts

Why use scripts?

- Readable
- Searchable
- Editable
- Reuseable
- Shareable
- Self-contained
- Can be combined in pipelines

3. Scripts

Parts of a script

docstring

```
"""
This script tells you how many pairs of shoes to buy
for a cow or an octopus.
"""
```

import statement(s) (if any)

```
#import module    #this script does not need any modules
```

function definitions

```
def count_legs(animal):
    if animal == "cow":
        number = 4
    elif animal == "octopus":
        number = 8
    return number

def calculate_shoe_order(animal, leg_count):
    pairs = int(leg_count / 2)
    shoe_order = ("For the "
                  + animal
                  + ", please order "
                  + str(pairs)
                  + " pairs of shoes.")
    return shoe_order
```

main function definition (contains function calls)

```
def main():
    n = count_legs("octopus")
    final_answer = calculate_shoe_order("octopus", n)
    print(final_answer)
```

main function call

```
if _name_ == "_main_":
    main()
```

3. Scripts

Docstrings

Python conventions for writing good docstrings are outlined in PEP 257.

For docstrings inside functions:

```
"""One line docstring."""
```

3. Scripts

Docstrings

For complicated functions and scripts:

```
"""
A short description of the script.
Any info that someone would need to run
your script.
"""
```

3. Scripts

Docstrings

Info that someone might need to know in the docstring:

- Any modules or other dependencies that they need to have
- An explanation of any parameters that have to be specified and any limitations to those parameters (e.g. This script takes two .txt files, each with one column of data.)
- A description of the output of the script (e.g. This script saves a .json dictionary with names as keys and addresses as values.)

3. Scripts

Importing modules

Modules are computer programs that contain functions.

Module functions are not included in Python's built-in program, and must be imported.

Many modules are included with Anaconda, others are included in Homebrew (Mac)

3. Scripts

Three ways to run scripts - IDE

Many IDEs provide a way to run an entire script that you have open in the IDE's text editor. This is convenient when you are writing the script. Usually the script will run in the Python shell that is open in the IDE (you will know this because any output from your script will appear in the shell). You should also test that your script works from the command line before you share it with others.

3. Scripts

Three ways to run scripts - Command line

You can run your script on the command line by typing:

```
python pathToMyScript/myScript.py
```

This allows you to run the script from within any directory.

3. Scripts

<u>Three ways to run scripts - In pieces</u>

You can also call individual functions from one script within another script. This is useful if you find yourself writing the same functions frequently.

When sharing your scripts, you must be sure that you also include all the scripts that you borrow functions from.

3. Scripts

if __name__ == "__main__"

When calling the main function in a script, it is best practice to use the following syntax:

```
if __name__ == "__main__":
    main()
```

3. Scripts

```
if __name__ == "__main__"
```

Using this syntax allows for the script to be both run as a full script and called as individual functions in another script.

If you do not use this, you will not be able to run individual functions in other scripts. Instead, when you import the script like a module, it will run the main function immediately.

3. Scripts

Practice

Let's get familiar with the sample script. Open sampleScript.py in your Text Editor.

Earlier, we learned that you should include a short docstring with each function. Take a minute to look at the script and figure out how it works. For each of the two functions (you do not need a docstring for the main function), write a short docstring that explains what the function does.

3. Scripts

<u>Calling your script from the command line</u>

In your command line shell, first change to the folder for this workshop.

Then, type:

```
python sampleScript.py
```

3. Scripts

Calling your script from the command line

The script was written to print something. When you call your script from the command line, it prints to what is referred to as ***standard output***. As a default, standard output is printed to the screen, but you can also redirect it to a file:

```
python sampleScript.py > sampleOutput.txt
```

3. Scripts

Calling a function from inside another script

If you used the correct syntax for calling the main function, you can now import your script into Python just like any other module. The imported script must be in the same folder as the script you are running, or, if you are importing it to the interactive Python shell, in your working directory.

3. Scripts

Calling a function from inside another script

In your Python shell:

```
import sampleScript
```

3. Scripts

Calling a function from inside another script

View the functions available in your module:

```
import sampleScript

dir(sampleScript)
```

3. Scripts

Calling a function from inside another script

Test your functions:

```
sampleScript.count_legs("cow")

sampleScript.count_legs("lion")

sampleScript.calculate_shoe_order("lion", 4)
```

3. Scripts

Calling a function from inside another script

You can also import single functions from a module:

```
from sampleScript import calculate_shoe_order

calculate_shoe_order("hyena", 4)

calculate_shoe_order("butterfly", 6)
```

3. Scripts

<u>Try to make your scripts as reusable and universal as possible</u>

Specific variables, like filenames and paths, should be assigned outside of individual functions whenever possible

3. Scripts

Try to make your scripts as reusable and universal as possible

BAD - this script only works on one specific file

```
def firstLine():
    with open("Documents/myFolder/myFile.txt", "r") as f:
        f1 = f.readline()
        print(f)
firstLine()
```

3. Scripts

Try to make your scripts as reusable and universal as possible

BETTER - you can change the function call

```
def firstLine(filename):
    with open(filename, "r") as f:
        f1 = f.readline()
        print(f)
firstLine("Documents/myFolder/myFile.txt")
```

3. Scripts

Try to make your scripts as reusable and universal as possible

When using filenames, always use the absolute path unless you have a good reason not to

3. Scripts

<u>Try to make your scripts as reusable and universal as possible</u>

BAD - you always have to run the script from the right directory

```
def firstLine(filename):
    with open(filename, "r") as f:
        f1 = f.readline()
        print(f)
firstLine("myFile.txt")
```

3. Scripts

Try to make your scripts as reusable and universal as possible

BETTER - you can run it from anywhere

```
def firstLine(filename):

    with open(filename, "r") as f:

        f1 = f.readline()

        print(f)

firstLine("~/Documents/myFolder/myFile.txt")
```

3. Scripts

<u>Try to make your scripts as reusable and universal as possible</u>

BEST - To make the script the most universal, you can even call specific variables from outside the script - i.e. you can change the variable frequently without changing the functionality of the code

4. sys module

<u>The sys module</u>

sys is a module that lets you pass information between the command line and your python script

This is different than os, which is a module that lets you call command line codes in Python

4. sys module

```
python script.py cow

cow
```

sys.argv lets you specify arguments on the command line that are then taken into your script and used

```
import sys

def say(word):
    print(word)


say(sys.argv[1])
```

4. sys module

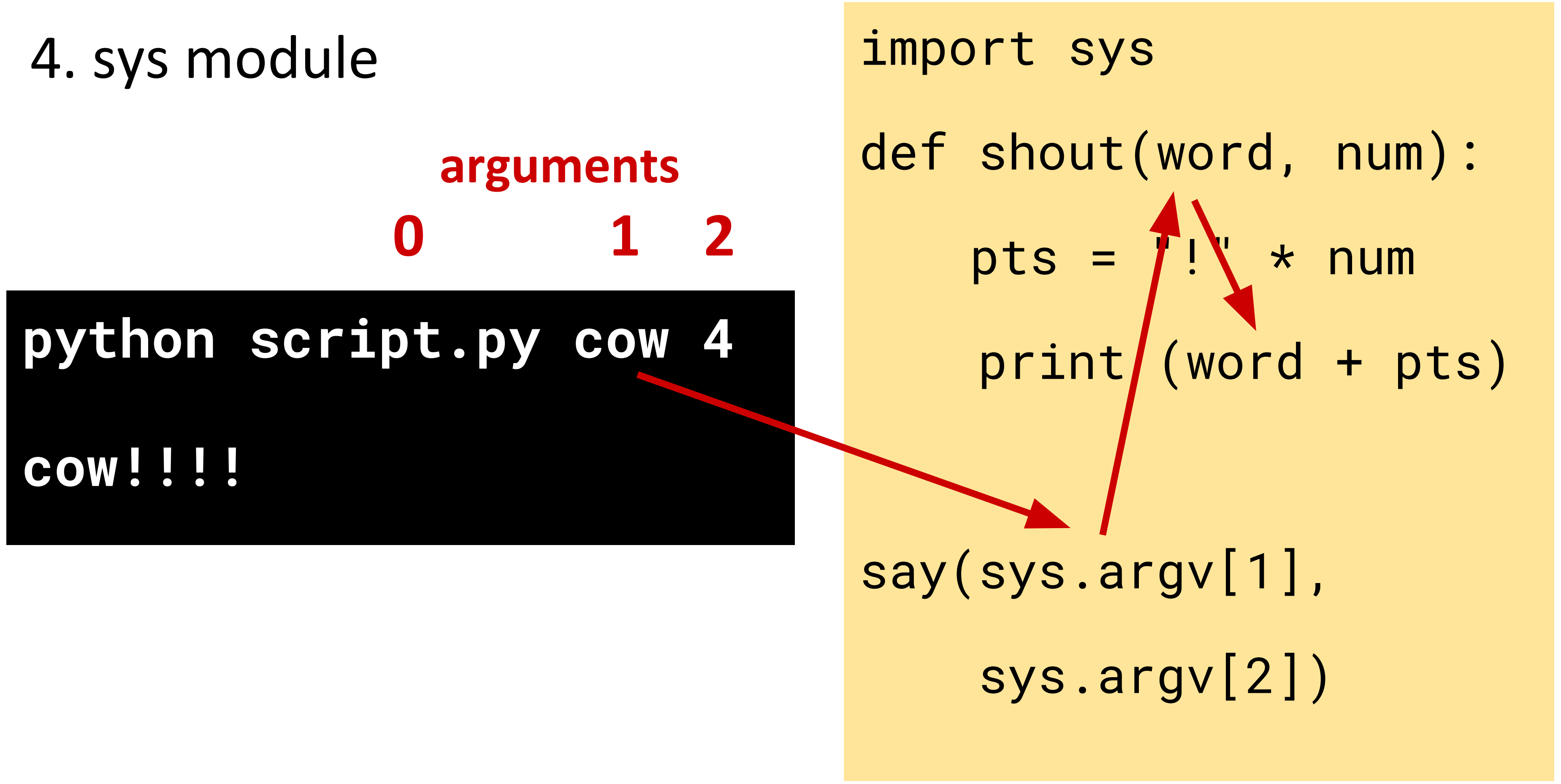
arguments

0 1 2

```
python script.py cow 4

cow!!!!
```

```
import sys

def shout(word, num):

    pts = "!" * num

    print (word + pts)


say(sys.argv[1],

    sys.argv[2])
```

4. sys module



```
python script.py cow 4

cow!!!!
```

```
import sys

def shout(word, num):

    pts = "!" * num

    print (word + pts)


say(sys.argv[1],

    sys.argv[2])
```

4. sys module

arguments

0 1 2

```
python script.py cow 4

cow!!!!
```

```
import sys

def shout(word, num):

    pts = "!" * num

    print (word + pts)


say(sys.argv[1],

    sys.argv[2])
```

4. sys module

sys.argv

If you use sys.argv, it is import to include info in the docstring about what the user should include on the command line. The argvs are also called **positional arguments** because they are arguments in which their position matters (e.g. in our last example the first argument had to be a string and the second had to be an integer)

4. sys module

sys.argv

There are also arguments called ***optional arguments***.

There are also more sophisticated modules for ***argument parsing***, like argparse.

4. sys module

<u>Other useful functions in sys</u>

sys.stdin and sys.stdout let you pass information between programs without having to write and open files - useful if you are creating large temporary data files that don't need to be permanently saved

4. sys module

Practice

Make our sampleScript.py more useful by incorporating sys.argv.

First, import the sys module.

Second, we've learned that the count_legs function is not useful as it only works for two animals. Let's remove it!

Third, rework the script to accept two arguments on the command line: an animal and a number of legs (instead of the very specific "octopus").

Finally, change the docstring to let users know which arguments are needed.

5. Error handling

5. Error handling

We are going to cover two different ways to handle errors that may come up when running your script or when running a function from your script.

Both of these require you to anticipate possible errors. You can do this by thinking about how data types might confuse the script, and by running the script with different arguments.

5. Error handling

There are many Python modules specifically for ***testing*** your functions and scripts to identify possible errors.

There are many articles with recommendations for testing for possible errors and handling errors.

We are only going to talk about two basic, but useful, methods for ***error handling***.

5. Error handling

try/except

The default for Python is to stop the script if an error occurs. Sometimes we don't want the script to stop, but we want to perform a different piece of code if a particular exception is raised, or we want the code to skip an item in a list but continue with the other items. We can use a try/except statement.

5. Error handling

<u>try/except</u>

Example: We want to cycle through a list of files and print the first line of each file. If a file is not found, we do not want the code to crash, but we want to continue with the other files in the list.

5. Error handling

try/except

Here we are calling a specific error, so other error types will cause the script to quit

```
file_list = ["sampleScript.py", "f.txt",
             "functionPractice.py"]
for file in file_list:
    try:
        with open(file, "r") as f:
            f1 = f.readline()
            print(f1)
    except FileNotFoundError:
        pass
```

5. Error handling

try/except

Don't forget PEP 20! Errors should never pass silently!

```
file_list = ["sampleScript.py", "f.txt",
             "functionPractice.py"]
for file in file_list:
    try:
        with open(file, "r") as f:
            f1 = f.readline()
            print(f1)
    except FileNotFoundError:
        print(file + " was not found.")
```

5. Error handling

<u>if/raise</u>

if/raise statements are good for two uses:

- if you want to raise a new error specific to your script that wouldn't raise a built-in error
- if you think the error could be so common that it's not worth attempting a try unless some condition is met

5. Error handling

<u>if/raise</u>

if you want to raise a new error specific to your script

```
def count_sentence(sentence):
    if sentence[0].islower():
        raise Exception("First letter of sentence must
                        be uppercase.")
    else:
        s = len(sentence.split(" "))
        print("The length of this sentence is {} words:
              {}".format(s, sentence))
```

5. Error handling

<u>if/raise</u>

if you think the error could be so common that it's not worth attempting a try unless some condition is met

```
def multiply_two_numbers(num1, num2):
    if type(num1) == str:
        raise Exception("num1 is string")
    elif type(num2) == str:
        raise Exception("num2 is string")
    else:
        product = num1 * num2
        print(product)
```

5. Error handling

Practice

Try running the sampleScript.py with different arguments that you think a user might try. Try it both as a script run on the command line, and as function called in your Python shell.

Write down any errors that you think might come up.

5. Error handling

<u>Practice</u>

Work as a group on your assigned error.

First, decide if this should be handled with try/except or if/raise (or some other solution)

Second, write the code to handle the error

6. Python etiquette 2

6. Python etiquette 2

Where you should NOT put spaces:

- immediately inside parentheses, brackets, and braces

NO:
```
my_string.split( "," )
```

YES:
```
my_list = [1, 4, 5]
```

6. Python etiquette 2

Where you should NOT put spaces:

- immediately before a comma, colon, or semicolon

NO: `print(list1 , list2)`

NO: `if number > 10 :`

YES: `with open(my_file, "r") as f:`

6. Python etiquette 2

Where you should NOT put spaces:

- between a function and the parenthesis that starts the argument list

  NO: `print (name, date)`

  YES: `str(2005)`

6. Python etiquette 2

Where you should NOT put spaces:

- between a variable and the square bracket that starts its index

NO: `my_dict [a_key]`

YES: `my_list[2]`

6. Python etiquette 2

Where you should NOT put spaces:

- inside empty parentheses, brackets, or braces

NO: `new_dict = { }`

YES: `new_list = []`

6. Python etiquette 2

Where you should NOT put spaces:

- inside a range

NO: `my_string[3 : 9]`

YES: `new_list[0:10]`

6. Python etiquette 2

Where you SHOULD put spaces:

- after a comma in a list or tuple

YES: `print(typeA, typeB, typeC)`

NO: `new_list = ["cat","dog","bicycle"]`

6. Python etiquette 2

Where you SHOULD put spaces:

- on either side of an operator, except if there are multiple operations...

YES: `sum_total = a + b`

NO: `if difference>c-b:`

6. Python etiquette 2

Where you SHOULD put spaces:

- on either side of an operator, except if there are multiple operations in which you can follow the order of operations

YES: `two_areas = W1*L1 + W2*L2`

NO: `shout = word1 + "!" * 3`

6. Python etiquette 2

Practice

Open up the file spacePractice.py in your Text Editor.

6. Python etiquette 2

<u>The 20th aphorism</u>

I like to think that the 20th is left for each person to fill in on their own with something that makes coding in Python easier for them.

6. Python etiquette 2

The 20th aphorism

Think before you type. Or write/draw/plan before you code.

Tips to become a better Python coder

- Use it under pressure
- Get a mentor
- Become a mentor
- Keep learning - RCS workshops, DataCamp, etc.
- Get help - RCS does consultations!
- If you don't know it, look it up
- Go back to PEP 8 and review it periodically
- (And, if you haven't yet, learn how to use `with` to open files, and learn list comprehension and dictionary comprehension)

6. Python etiquette 2

<u>Practice</u>

Yesterday we wrote a function to calculate the acreage of a rectangle. You are going to write a script that takes that function to the next level.

6. Python etiquette 2

Practice

You will be given:

1. a .txt file of various measurements of area (square km, acre, etc.) and the equivalent number of square feet (areas.txt)
2. a .txt file with several rectangular plots of land and their lengths and widths in feet (plots.txt)

Take a look at these files to see how they are structured

6. Python etiquette 2

With pencil and paper, outline how you would organize a script.

The script should accept two arguments on the command line: the name of the plot and the measurement to be returned

The script should print out the name of the plot and the correct area based on the measurement unit requested. For example: Lot 45 is .8 square kilometers

Specify what functions you will need and write the main function.