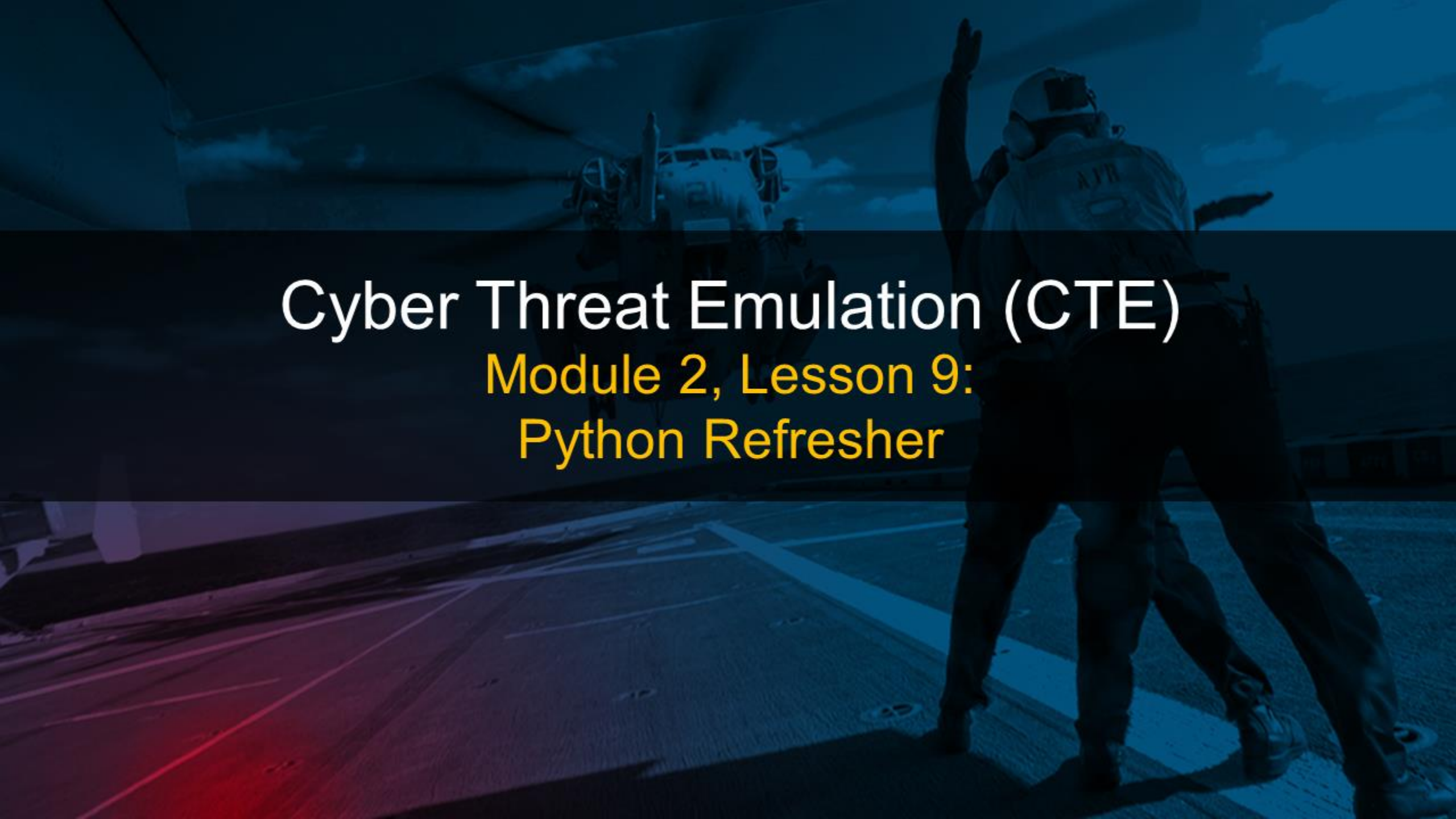


Cyber Threat Emulation (CTE)

Module 2, Lesson 9:
Python Refresher

Course Objectives

After completing this course, students will be able to:

- Summarize the CTE squad's responsibilities, objectives, and deliverables from each CPT stage
- Analyze threat information
- Develop a Threat Emulation Plan (TEP)
- Generate mitigative and preemptive recommendations for local defenders
- Develop mission reporting
- Conduct participative operations
- Conduct reconnaissance
- Analyze network logs for offensive and defensive measures

Course Objectives (Continued)

Students will also be able to:

- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan non-participative operations using commonly used tools, techniques and procedures (TTPs)

Module 2: Threat Emulation (Objectives)

- Conduct reconnaissance
- Generate mission reports from non-participative operations
- Plan a non-participative operation using social engineering
- Plan a non-participative operation using Metasploit
- Analyze network logs for offensive and defensive measures
- Analyze network traffic and tunneling protocols for offensive and defensive measures
- Plan a non-participative operation using Python
- Develop fuzzing scripts
- Develop buffer overflow exploits

Module 2 – Lesson 9: Python Refresher (Objectives)

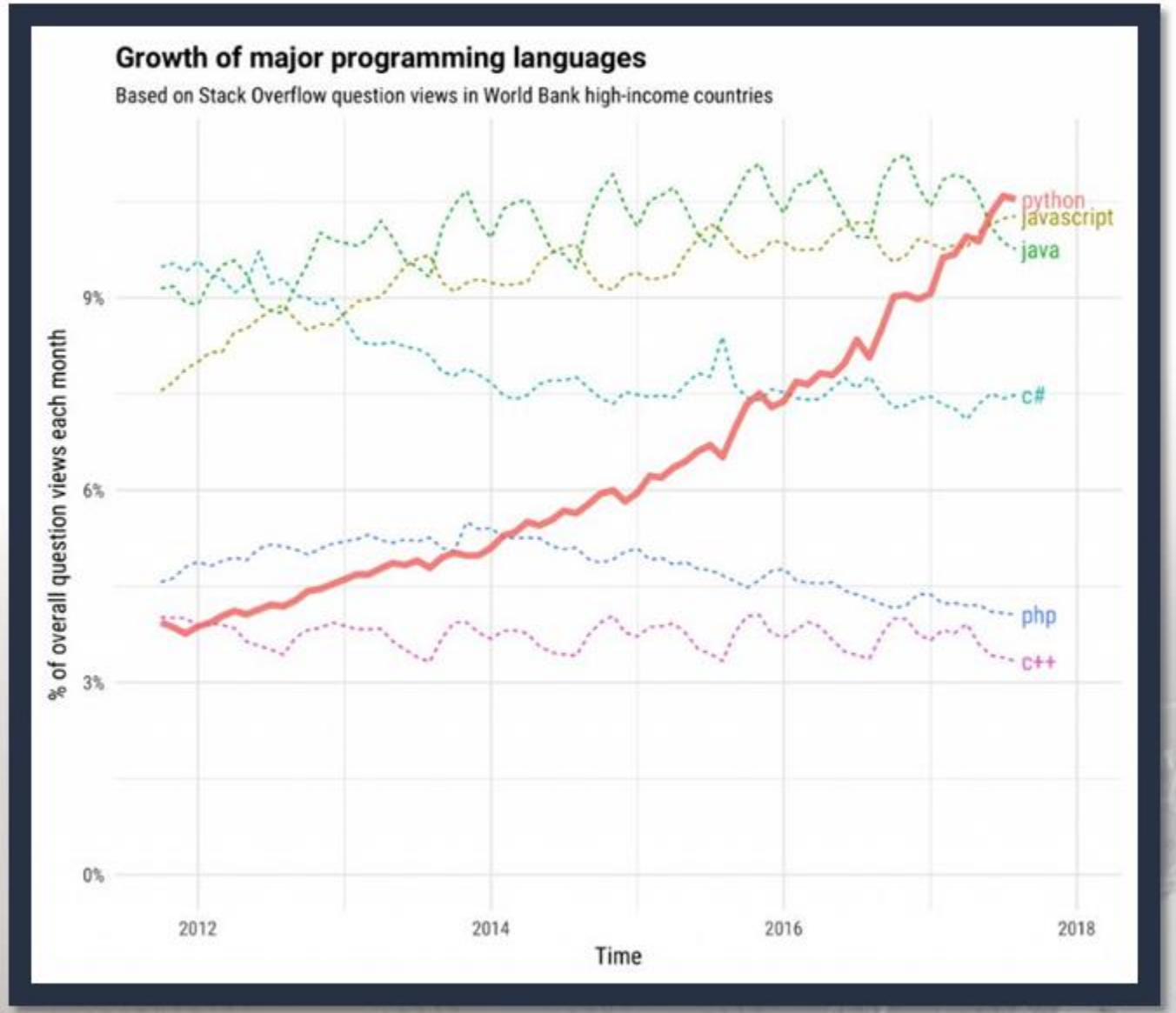
- Manipulate variables, strings, lists, dictionaries, conditionals, loops, and functions in Python
- Create variables, strings, lists, dictionaries, conditionals, loops, and functions in Python

Python History

- 1989 – Python was created
- 1991 – Python 1
- 2000 – Python 2
  - Most applications still use Python 2
  - Python 2 has features that are not forward compatible
  - End of life projected for 2020
- 2008 – Python 3

Why Python?

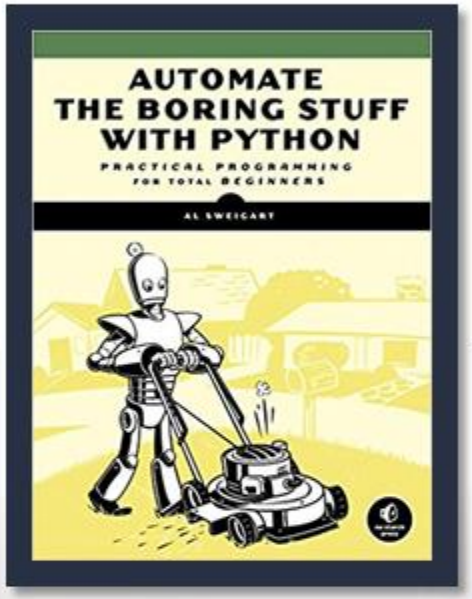
- Popularity
- Availability



Why Python? (continued)

Many benefits:

- Open source
- Easy to learn
- Easy to read
- Many modules are available
- Runs on multiple platforms
- Suitable for everyday tasks
- Web development
- Data science
- Machine learning
- Mine Twitter data
- Create games



What is Python?

Python is interpreted

- Similar to PERL and PHP
- No compiling is needed like in C or C++
- Python is processed at runtime by an interactive interpreter

Python is interactive

- Type straight into an interactive CLI Python interpreter

Python is Object-Oriented

- Python supports Object-Oriented programming that encapsulates code within an object

What is Python: Compiled vs Interpreted Languages

Compiled Language

```
#include<iostream>
using namespace std;

int main() {
int number, reverse = 0;
 cout<<"Input a Number to Reverse:  ";
 cin>> number;

   for( ; number!= 0 ; )
   {
      reverse = reverse * 10;
      reverse = reverse + number%10;
      number = number/10;
```

Compiler

The script goes through a compiler like gcc.

Binary

The compiler compiles the code into binary for the machine to use.

Interpreted Language

```
import collections
import requests

MovieResult = collections.namedtuple(
    'MovieResult',
    "imdb_code,title,duration,director,year,ratin


def find_movies(search_text):

    if not search_text or not search_text.strip()
        raise ValueError("Search text is required

    url = 'http://movie_service.talkpython.fm/api
```

Interpreter

The interpreter is an executable running on the machine and runs scripts written in an interpreted language.

Interactive Interpreter or REPL (Read Evaluate Print Loop)

Interactive Interpreter or REPL (Read Evaluate Print Loop)

#type python3 to launch the interactive interpreter

#the interpreter **reads** the input which is 40 + 2

#the interpreter **evaluates** the input

#the interpreter will **print** the response

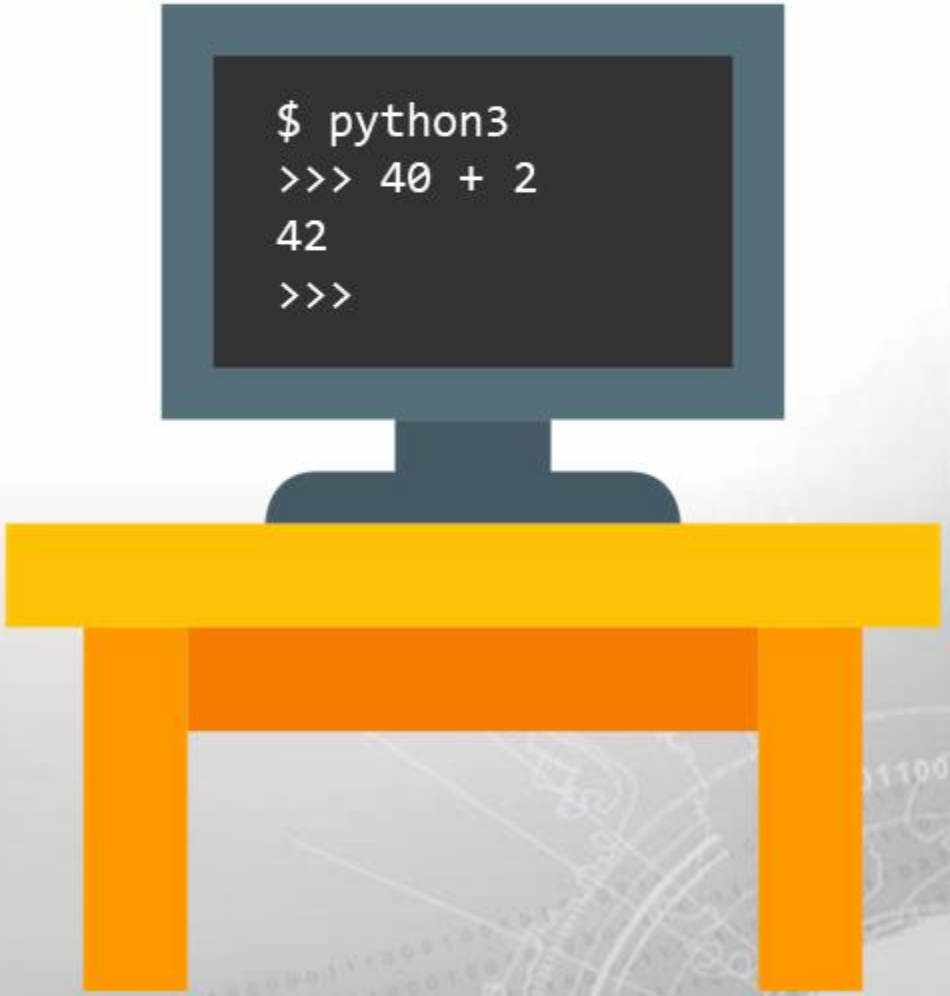
#the interpreter will **loop** back to the beginning and wait for the next thing to read

R

E

P

L



Editors & Integrated Development Environments (IDEs)

Windows

- Notepad++
- Visual Studio
- PyScripter
- PyCharm
- Atom
- Emacs
- IDLE

Linux

- Ninja-IDE
- PyCharm
- Gedit
- VIM
- Atom
- Nano
- IDLE

OS X

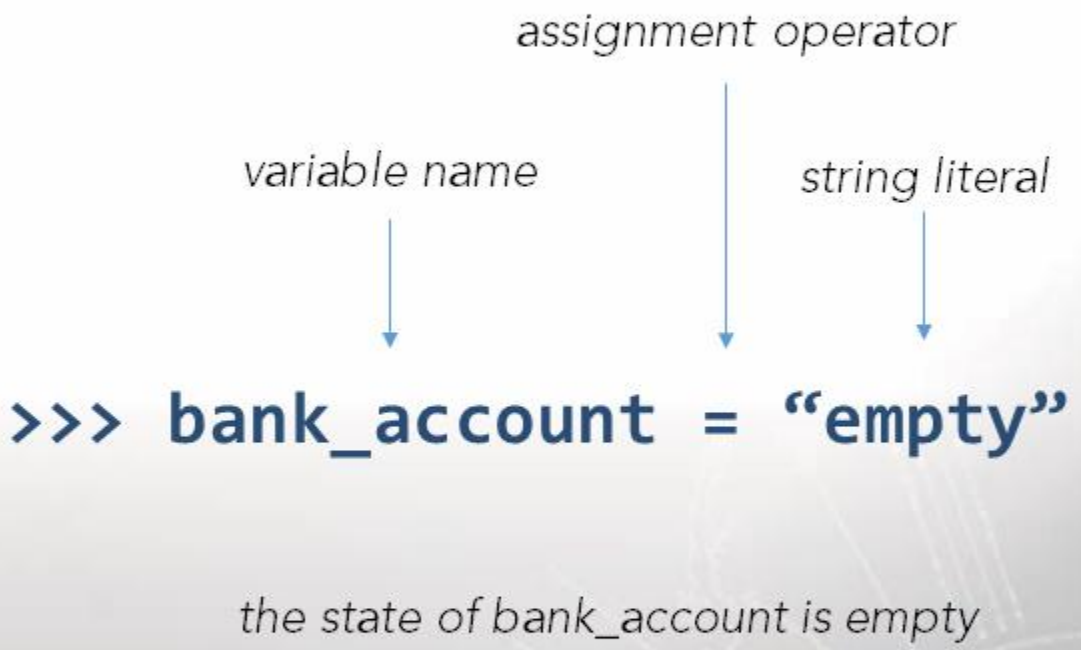
- Xcode
- EditXT
- PyCharm
- Pydev
- TextMate
- IDLE

Bonus Activity

- An “always-on” exercise is available to you throughout the Python portion of this course.

Variables: Creating

- Variables are the basic building blocks in programming
- Everything is seen as an object
- Variables keep track of state
  - State: id, type, and value associated with the object



Variables: Naming Convention

- Pythonic code and Pythonistas
- Do not use keywords as variable names!
  - Results in a SyntaxError
- Use an underscore to separate words
- Do not start with numbers
- Keep lowercase

Keywords

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with',
'yield']
```

Standards

- https://www.python.org/dev/peps/
- https://pep8.org/

Help Function

Type '**help()**' to access Python 3.6's help utility!

Variables: Naming Convention

- Python will let you use built-ins as variables, but you should avoid using them.

Commonly Used Built-In Functions	
dict	id
list	min
max	open
range	str
sum	type

- Fight the temptation to use these as variable names!

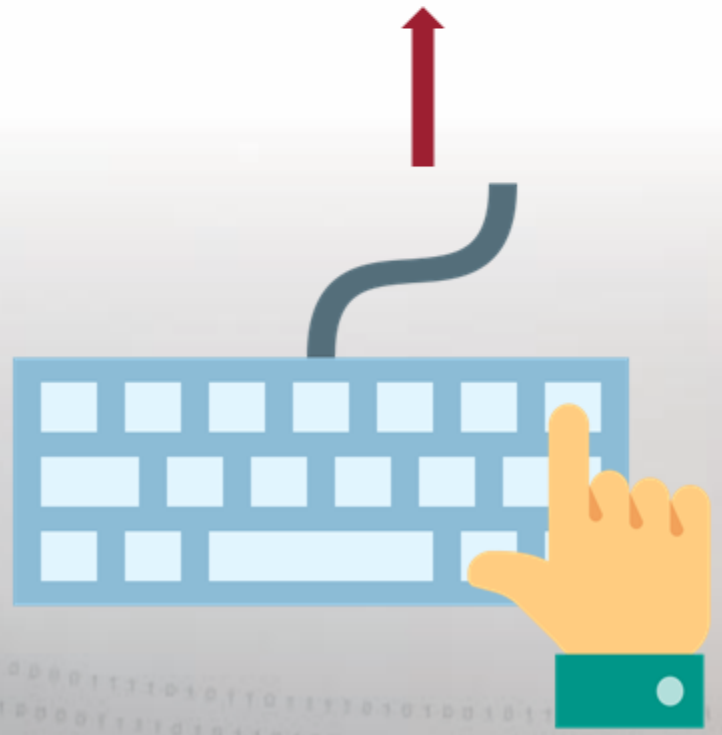
Built-in Functions

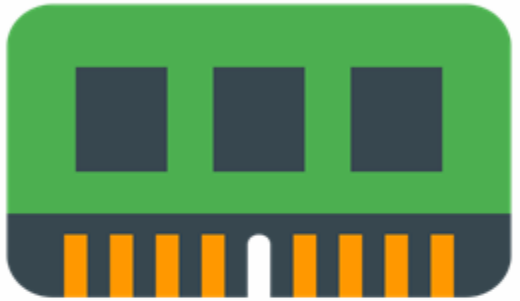
```
>>> dir(__builtins__)
['ArithmeticError',
'AssertionError',
'AttributeError', 'BaseException',
'BlockingIOError',
'BrokenPipeError',
'BufferError', 'BytesWarning',
'ChildProcessError',
'ConnectionAbortedError',
'ConnectionError',
'ConnectionRefusedError',
'ConnectionResetError',
'DeprecationWarning', 'EOFError',
'Ellipsis', 'EnvironmentError',

<<<<<THIS LIST CONTINUES>>>>>>
```

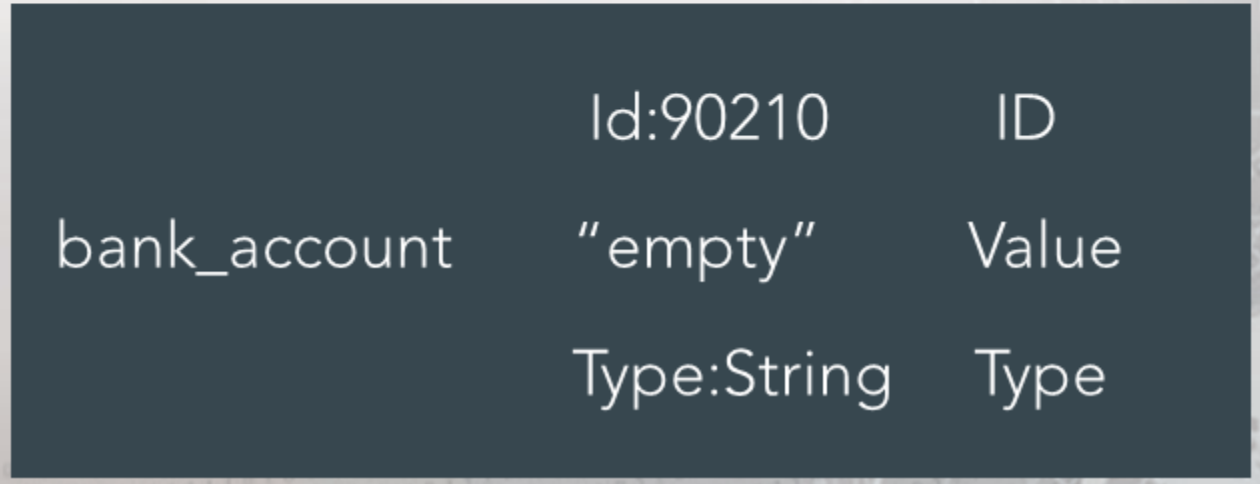
Variables: value

- value: the data that the object holds

```
>>> bank_account = "empty"
```





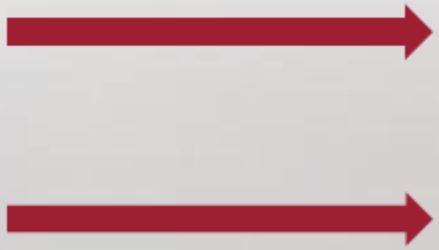


Variables: id

- id: A unique ID for the object; its location in memory

Explain what is occurring with these variables?

Notice anything?



id()

```
>>>bank_account = "empty"
>>>id(bank_account)
139837910943032
>>>house = bank_account
>>>id(house)
139837910943032
>>>id(bank_account)
139837910943032
```

Variables: type

- type: the class of the object
- Reveal an object's type by passing it through type():

type()

```
>>>type(name)
<class 'str'>
```

Objects and Types

Objects	Types
String	str
Integer	int
Floating Point	float
List	list
Dictionary	dict
Tuple	tuple
function	function
type	type
subclass of class	class
Built-in function	builtin_function_or_method

Writing and Reading

```
>>> print('Nobody likes you')      #print will write to standard out
Nobody likes you
>>> print('We are', 2, 'wild and crazy guys')  #print statement w/ strings and numbers
We are 2 wild and crazy guys
>>> print('The', 92, 'Dream Team')
The 92 Dream Team
>>> my_age = input("What's my age again?") #waits for input and stores answer in variable my_age
What’s my age again? 23
>>> my_age
'23'
```

Writing and Reading

```
>>> my_age = input('Enter a number:')
23
>>> real_age = input('Enter another:')
42
>>> type(my_age)
<class 'str'>
>>> my_age + real_age
'2342'
>>> int(my_age) + int(real_age)
65
```

Input waits for input and stores the input in the variables **my_age** and **real_age**

The variable **my_age** is a number, but type() reveals that it is stored as a string

int() converts **my_age** and **real_age** from strings into integers, letting you add them

Mutability

Mutable

Mutable objects can change their value; they can alter their state but their identity stays the same.

Mutable Types:

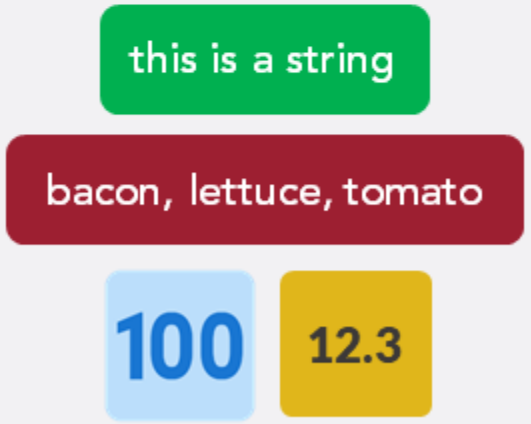
- Dictionaries
- Lists

Immutable

Immutable objects do not allow their value to be changed; once their value changes their ID changes as well.

Immutable Types:

- Strings
- Tuples
- Integers
- Floats



Numbers

Integers and Floats

```
>>> type(1)
<class 'int'>
>>> type(2.0)
<class 'float'>
```

Addition

```
>>> 40 + 2
42
>>> result = _
>>> result
42
```

Explicit Conversion

```
>>> int(2.3)
2
>>> float(3)
3.0
```

Coercion Between Strings and Numbers

```
>>> print('number: %s' % 42)
number: 42
>>> 'Pizza!' * 2
'Pizza!Pizza!'
>>> '4' * 2
'44'
```

Numbers

Subtraction

```
>>> 2 - 44
-42
>>> .25 - 0.2
0.04999999999999999
>>> 5 - .8
4.2
```

Division

```
>>> 84/2
42.0
>>> 3 / 4
0.75
>>> 3 // 4   #floor division
0
```

Power

```
>>> 4 ** 2
16
>>> 10 ** 100
10000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
0000000000000000
```

Multiplication

```
>>> 21 * 2
42
>>> 4 * .3
1.2
```

Modulo

```
>>> 4 % 3
1
>>> 3 % 2 # odd if 1 is result
1
>>> 4 % 2 # even if 0 is result
0
```

Numbers

- Python follows the traditional PEMDAS order of operations.

Order of Operations

`>>> 4 + 2 * 3`
`10`
`>>> (4 + 2) * 3`
`18`

("Please excuse my dear Aunt Sally")

PEMDAS

- P — Parenthesis
- E — Exponent
- MD — Multiplication & Division
- AS — Addition & Subtraction

Strings

- Strings are immutable objects that hold character data
  - Could be a single character, word, paragraph, multiple paragraphs, even zero characters

- Denoted by wrapping with:
  - ' (single quotes)
  - " (double quotes)
  - """(triple doubles)
    - Usually used as a docstring
  - ''' (triple singles).

Single Quotes

```
>>> first = 'never gonna'
```

Double Quotes

```
>>> second = "give you up"
```

Triple Doubles

```
>>> triples = """Never gonna let
... you down. Never gonna run
... around"""
```

Strings: Escaping Characters

Escape Sequence	Output
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\b`	ASCII Backspace
`\n`	Newline
`\t`	Tab
`\u12af`	Unicode 16 bit
`\U12af89bc`	Unicode 32 bit
`\N{SNAKE}`	Unicode character
`\o84`	Octal character
`\xFF`	Hex character

Strings: Avoid Escaping with r

- Have r precede a string to turn the string into a raw string.
- This is usually done with regular expressions and in Windows paths where the backslash is the delimiter.

r

```
>>> slasher = r'\tNorman Bates \n\t  \\Motel\\'
>>> print(slasher)
\tNorman Bates \n\t  \\Motel\\
>>>
>>> slashless = '\tNorman Bates \n\t
\\Motel\\'
>>> print(slashless)
        Norman Bates
          \Motel\
```

Strings: Formatting Strings

- Use the .format method

Using .format

```
>>> 'child: {}'.format('Beyonce')
'child: Beyonce'
>>>
>>> 'child: {name}'.format(name='Kelly')
'child: Kelly'
>>>
>>> 'child:
{[child]}'.format({'child':'Michelle'})
'child: Michelle'
>>>
>>> 'Last: {2},  First:
{0}'.format('Beyonce','Kelly','Michelle')
'Last: Michelle,  First: Beyonce'
```

'Kelly' is the value of the name.

'Michelle' is an indexed item in the 'child' dictionary.

'Beyonce', is at position 0, the second, 'Kelly', is position 1, and 'Michelle' is at 2.

Strings: F-Strings

- F-strings introduced in Python 3.6
- Allow for the inclusion of code in placeholders
  - Placeholders can contain function calls, method calls, or any arbitrary code
- Potential for exploitation
  - Reliance on input from outside of the original code

Using f-strings

```
>>> name = 'Marshall'
>>> f'Hi. My name is {name}'
'Hi. My name is Marshall’
>>> f'Hi. My name is {name.lower()}'
'Hi. My name is marshall'
```

Strings and Methods

- Because everything is seen as an object, each of the objects have methods.
- Invoking dir on an object will show the methods associated with the object.
- STRINGMETHODS entry in the help section has documentation and examples.

Common String Methods

.endswith	.find
.format	.join
.lower	.startswith
.strip	.upper

Some Examples

```
>>> print('first: {}, last:
{}'.format('Jack','Burton'))
first: Jack, last: Burton
>>> phrase = "It's all in the reflexes."
>>> phrase.find('all')
5
>>> phrase.find('reflex')
16
>>> 'lo pan'.upper()
'LO PAN'
```

Strings and Methods: dir

- dir() returns the attributes of objects

Using dir

```
>>> dir(phrase)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Strings and Methods: help

- help() brings up Python's help utility

Using help

```
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the
Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using
Python modules.  To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords",
"symbols", or "topics".  Each module also comes with a one-line summary of what it does; to list
the modules whose name or summary contain a given string such as "spam", type "modules spam".
```

Strings and Methods: help

- help() also provides documentation for a method, module, class or function.

Using help

```
>>> help(name.upper)

Help on built-in function upper:
upper(...) method of builtins.str instance
    S.upper() -> str

 Return a copy of S converted to uppercase.
(END)
```

```
>>> name = 'Marshall'
>>> f'Hi. My name is {name.upper()}'
'Hi. My name is MARSHALL'
```

pdb

- Built-in Python debugger

Using pdb

```
>>> pdb.set_trace()
--Return--
> <stdin>(1)<module>()->None
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF     c          d        h         list      q        rv       undisplay
a       cl         debug    help      ll        quit     s        unt
alias   clear      disable  ignore    longlist  r        source   until
args    commands   display  interact  n         restart  step     up
b       condition  down     j         next      return   tbreak   w
break   cont       enable   jump      p         retval   u        whatis
bt      continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb)
```

Lists

- Mutable
- Holds a list of objects
- May hold any type of item, but it's good practice to hold a single type of item

Create An Empty List
(Two different ways)

```
>>> movies = list()
>>> movies = []
```

Invoke the list class

Note the empty square brackets

Create A Populated List

```
>>> movies = ['Ferris Beuller\'s Day Off', 'Gladiator', 'Golden Child’]
>>> print(movies)
['Ferris Beuller's Day Off', 'Gladiator', 'Golden Child']
```

Lists

- Use methods to modify lists
  - E.g., .append, .remove, .sort

.append

```
>>> movies.append('Big Trouble in Little China')
```

.remove

```
>>> movies.remove('Grease 2')
```

.sort

```
>>> movies
["Ferris Beuller's Day Off", 'Gladiator', 'Golden Child', 'Big Trouble in Little China']
>>> movies.sort()
>>> movies
['Big Trouble in Little China', "Ferris Beuller's Day Off", 'Gladiator', 'Golden Child']
```

Tuples

- Immutable
- Cannot use .append or .remove methods
- Elements are enclosed in parentheses
- Boolean values
  - An empty tuple is false
  - Tuples with at least one item are true

Create A Tuple With 0 Items

```
>>> empty = tuple()
>>> empty
()

---------- OR ----------

>>> empty = ()
>>> empty
()
```

Create A Tuple With 1 Item In It

```
>>> one = tuple([1])
>>> one
(1,)
>>> one = (1,)  #pythonic
>>> one
(1,)
>>> one = 1,
>>> one
(1,)
```

Create A Tuple With Multiple

```
>>> amigos = tuple(['Chase','Short','Martin'])
>>> amigos
('Chase', 'Short', 'Martin')
>>> amigos = 'Chase','Short','Martin'
>>> amigos
('Chase', 'Short', 'Martin')
>>> amigos = ('Chase','Short','Martin')  #pythonic
>>> amigos
('Chase', 'Short', 'Martin')
```

Sets

- Immutable
- Unordered collection
- Cannot contain duplicates
- Does not care about order
- Good for removing duplicates and checking membership

A set of numbers

```
>>> some_numbers =
{0,1,2,3,4,5,6,7,8,9,200,11,5,2,1,88}
>>> some_numbers
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 200, 11, 88}
```

What happened?

Create set to compare

```
>>> odd = {1, 3, 5, 7, 9,11}
>>> 9 in odd
True
```

Find the difference

```
>>> new_set = some_numbers - odd
>>> new_set
{0, 2, 4, 6, 8, 200, 88}
```

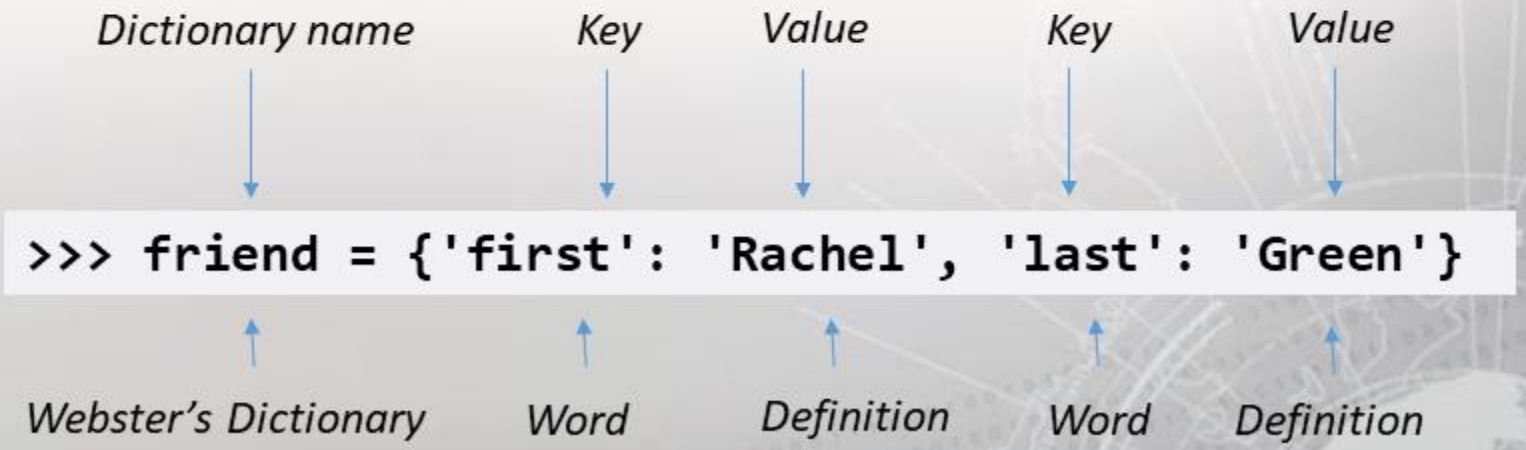
Find the intersection

```
>>> intersection = some_numbers & odd
>>> intersection
{1, 3, 5, 7, 9,11}
```

Dictionaries

- Link a key to a value
- Mutable
- Do not copy the variable
- Increase reference count to the variable

A Python dictionary contains one or more values known as keys [i.e. Baltimore], and each key is tied to a value [i.e. Orioles]. Multiple key-value pairs can be stored in a dictionary.



Dictionaries

- Manipulating dictionaries
- No duplicate keys; assigning a value replaces the previous one

Create

```
>>> friend = {'first': 'Rachel', 'last': 'Green'}
>>> friend
{'first': 'Rachel', 'last': 'Green’}
```

Change Value

```
>>> friend['last'] = 'Geller'
>>> friend
{'first': 'Rachel', 'last': 'Geller’}
```

Add Key-Value Pair

```
>>> friend['status']= 'married'
>>> friend
{'first': 'Rachel', 'last': 'Green', 'status': 'married'}
```

Exercise: Python Refresher I

Objectives

After completing this exercise, students will be able to:

- Create variables, strings, lists, tuples, sets and dictionaries in Python
- Manipulate variables, strings, lists, tuples, sets and dictionaries in Python

Duration

This exercise will take approximately **2.5** hours to complete, with **30-45 minutes** to review answers.

Debrief: Python Refresher I – Variables

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Variables

Specific Questions

- How can a variable type be determined in a Python interpreter?
- How can `port_list` be utilized in a script?
- Which of the following are correct naming conventions for variables?

```
123abc = 7
something with spaces = 42
pass = 42
my_variable = 42
my_1_kabob = 42
break = 42
```

Debrief: Python Refresher I – Numbers

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Numbers

Specific Questions

- What is the output of the following code?

```
4 + 2 * 3
```

- What is the output of the following code?

```
1 + 3.0 + 2
```

- What is the output of the following code?

```
(4 + 2) * 3
```

- What is the output of the following code?

```
42
42.0
4/2
```

Debrief: Python Refresher I – Numbers

Specific Questions

- What code needs to be entered into the interpreter to figure out the 2 to tenth power?
- What code needs to be entered into the interpreter to turn 57.2 to a type int?
- What code needs to be entered into the interpreter to turn 57.2 to a type str?
- What is the output of the following code?

```
42%(4/2)
```

Debrief: Python Refresher I – Numbers

Specific Questions

- You tracked how many hours during a work week you played video games. On Monday you played 1 hour, Tuesday 3.5 hours, Wednesday 1.2 hours, Thursday 3.3 hours, and Friday 6 hours. What is the equation you would enter into the Python interpreter to determine the average number of hours you played video games?
- Dividing an integer by a float will result with what?
  a. An integer
  b. A float
  c. A SyntaxError

Debrief: Python Refresher I – Numbers

Specific Questions

- Which is the output of the following code?

```
width = 5
height = 10
width * height
```

a. 50
b. widthwidthheightheight
c. 510

- What is the output of the following code?

```
'Wepa!' * 2
```

Debrief: Python Refresher I – Numbers

Specific Questions

- What is the output of the following code?

```
'4' * 2
```

- What is the output of the following code?

```
keaton = 'beetlejuice'
print(keaton * 3)
```

- What is the output of the following code?

```
>>> 40 + 2
42
>>> result = _
>>> result
```

Debrief: Python Refresher I – Strings

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Strings

Specific Questions

- How can a variable classified as an integer be converted to a string?
- What is the output of the following code?

```
x = 55.5
x = 'string or integer'
print(x + '?')
```

Debrief: Python Refresher I – Strings

Specific Questions

- Fill in the blanks to get the output:

```
1 fish 2 fish
```

```
x =
y =
f =
print(_____________)
```

- How can `<string>.format()` be used?

Debrief: Python Refresher I – Strings

Specific Questions

- Why did `print(jhopper.find("are"))` return a value of 9?
- On line 62 of the walk-through, the result of `my_docstring` being printed out had characters that were not originally entered in the docstring. What is the purpose of those characters? Without making any modifications to the value of the variable, how can `my_docstring` be printed without those characters displayed?
- What do `lstrip()` and `rstrip()` do?

Debrief: Python Refresher I – Lists

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Lists

Specific Questions

- Where is an item placed when appended to a list?
- What occurs when a list is passed through `len()`?
- What occurs when a list that contains both integers and strings is sorted without any parameters?
- In Part 2 of the walk-through on line 30, explain what happened when `port_list[0:15:3]` was run?

Debrief: Python Refresher I – Lists

Specific Questions

- On line 37 through 39 in Part 2, after running `port_list.append(8080)` the following output for `print()` does not show a increment in total ports scanned. Why does it still show 15 total ports?
- In Part 3 a variable `some_nums` was set to the value of `range(7)`. On line 17 in Part 3, `some_nums` returned numbers 0 through 6. Why was 7 not included?
- In a follow up to question 6, why did `some_nums` return a list that started with 0?

Debrief: Python Refresher I – Lists

Specific Questions

- Why did result of line 19 kick back an error but line 24 did not?
- What occurs when you reference an index that does not exist in a list?
- Write the syntax that would print `douglas` from the following list.

```
foo = ['jordan','douglas','jackson']
```

- How many items are in the following list:

```
foo = [2,]
```

Debrief: Python Refresher I – Lists

Specific Questions

- Which lines of code will cause an error? (more than one answer possible)

```
bar = [1307,'concourse',[3],'antoniovas', 42]
```

```
print(bar[2])
print(bar[1])
print(bar[1,3])
print(bar[1],bar[3])
print(bar[5])
```

```
print(bar[1],[3])
print(bar[-2])
print(bar[1:2])
print(bar[0:6])
print(bar[0:0])
```

Debrief: Python Refresher I – Lists

Specific Questions

- What is the result of the following code?

```
foo = ['rocky','skye','rubble','marshall','chase']
foo[1] = foo[3]
print(foo[1])
```

- What is the result of this code and why?

```
nums = list(range(7,11))
print(len(nums))
```

Debrief: Python Refresher I – Lists

Specific Questions

- What is the result of this code and why?

```
print(range(20) == range(0,20))
```

- What is the result of this code and why?

```
nums = list(range(42))
print(nums[4])
```

- What is the result of this code and why?

```
nums = list(range(42))
print(nums[-2])
```

Debrief: Python Refresher I – Lists

Specific Questions

- What is the result of this code and why?

```
nums = list(range(42))
print(nums[42])
```

- What is the result of this code and why?

```
nums = list(range(3,15,3))
print(nums[2])
```

Debrief: Python Refresher I – Tuples & Sets (Solutions)

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Tuples & Sets (Solutions)

Specific Questions

- Which syntax returns a tuple?

```
pb = (3,)
```

```
pb = (3)
```

```
pb = ([3])
```

```
pb = ([3,])
```

```
pb = ('j',)
```

```
pb = tuple()
```

```
pb = (['3',])
```

```
pb = (['3'])
```

```
pb = '3','7'
```

```
pb = '3'
```

```
pb = {}
```

```
pb = ((4*2),'j',37)
```

```
pb = (4*2,'j',37)
```

```
pb = ()
```

Debrief: Python Refresher I – Tuples & Sets (Solutions)

Specific Questions

- What is the difference between a list and a tuple?
- What is the result of the following syntax?

```
pb = ('jiff','peter pan')
pb = ('jiff','peter pan','skippy')
pb.append('smuckers')
```

- What is the result of the following syntax?

```
china = {1,1,1}
print(china)
```

Debrief: Python Refresher I – Tuples & Sets (Solutions)

Specific Questions

- What is the value of `porcelain` after the following syntax?

```
china_tea =
{1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,
3,3,3,3,3,3,3,3,3,3}
english_tea =
{4,4,4,4,4,4,4,4,4,4,4,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1}
porcelain = china_tea - english_tea
```

Debrief: Python Refresher I – Tuples & Sets (Solutions)

Specific Questions

- What is the value of the variable `letters`?

```
phoenician =
{'a','a','b','b','b','c','c','c','d','e','f','
g','h','i','j','k','l','m','n','o','p','q','r'
,'s','t','u','v','w','x','y','z'}
alphabet = {'a','b','c'}
letters = phoenician & alphabet
```

Debrief: Python Refresher I – Dictionaries (Solutions)

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher I – Dictionaries

Specific Questions

- Is a dictionary mutable?
- Can a key in a dictionary hold two different values? What would occur if attempted?
- What can the `in` statement be used for?
- Can a value in a dictionary be held by different keys?

Conditionals

- If *this* is true then do *this*
- Determine the truthiness of something
- if statements
- else statements

Operator	Meaning	Example
==	Equals	a == b
!=	Not equals	a != b
<	Less than	a < b
<=	Less than or equal to	a <= b
>	Greater than	a > b
>=	Greater than or equal to	a >= b
is	Identical object	a is a
is not	Not identical object	a is not a

Conditionals

```
>>> natalie = 8
>>> natalie == 8
True
>>> natalie == natalie
True
>>> natalie != 7
True
>>> natalie > 9
False
```

colon starts code block

```
>>> if natalie == 8:
...     print('her age')
... else:
...     print('not her age')
...
her age
```

white space matters

Code Blocks

- Denoted with a colon
- Denoted with indentation
- Whitespace matters

```
>>> if natalie > 8:
...     print('too old')
... elif natalie < 8:
...     print('too young')
... else:
...     print('correct')
...
correct
```

Iteration

- Being able to iterate (repeat) through objects
- __iter__()

Iterate through a tuple

```
>>> throne_seeker = iter(houses)
>>> print(next(throne_seeker))
stark
>>> print(next(throne_seeker))
lannister
>>> print(next(throne_seeker))
baratheon
>>> print(next(throne_seeker))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Using for to loop

```
>>> houses = ("stark","lannister","baratheon")
>>> for house in houses:
...       print(house)
...
stark
lannister
baratheon
>>>
```

Loop with an index

```
>>> houses = ["stark","lannister","baratheon"]
>>> for index in range(len(houses)):
...     print(index, houses[index])
...
0 stark
1 lannister
2 baratheon
```

Iteration: for loops

```
houses = ("stark","lannister","baratheon")
for house in houses:
    print(house)
```

The variable house will first point to the value stark and then iterate through each proceeding value.

house *becomes the variable. This word is arbitrary, but often will be related to the item being iterated through.*

The ***print*** *command will sequentially reference each item in the range.*

When the loop completes, Python doesn't clean up after itself and the last value passed through the variable remains:

```
>>> print(house)
baratheon
```

Iteration: in statement

- The in statement can be used to check membership

in statement

```
>>> houses = ("stark","lannister","baratheon")
>>> 'stark' in houses
True
>>> 'tyrell' in houses
False
```

Iteration: while statement

- loops over block of code while condition holds
- the condition can evaluate to True or False

```
>>> i = 0
>>> while i < 6:
. . .        print(i)
. . .        i += 1
0
1
2
3
4
5
>>> print(i)
6
```

What is happening in this example?

Iteration: break

- To break out of a loop, use the keyword break

Using break in a while loop

```
i=0
while i < 6:
    print(i)
    if i == 3:
        break
    i +=1
```

Using break in a for loop

```
for i in range(9):
    if i > 3:
        break
    print(i)

0
1
2
3
```

Functions

The key components of a function are:

- the def keyword (def defines that a function is being created)
- a function name (PEP8 says keep it lowercase, use underscores between words, can't be a key word, don't start with a number, don't override built-ins)
- function parameters parentheses (Any input parameters or arguments should be placed within these parentheses)
- a colon (:)
- code blocks (white space matters)
- a docstring (optional)
- a return statement (implied if not explicit)

Functions

def indicates a function is being defined

White space always matters!

my_func is the name of the function

Using triple quotes to begin and end a docstring

() parameters passed through the function

: (the colon) indicates the beginning of a code block

This the code that will be executed when the function is called

```
>>> def my_func():
...     """
...     This is a docstring
...     """
...     print("my function")
...
>>> my_func()
my function
```

Functions: Docstrings and help()

- After a docstring is written in a function, help() can be used to see what the author provided for the docstring of the function.

```
>>> def my_func():
...     """
...     This is a docstring that
...     will describe what my_func
...     does, but the docstring is
...     optional
...     """
...
>>> help(my_func)
```

```
Help on function my_func in module
__main__:

my_func()
    This is a docstring that
    will describe what my_func
    does, but the docstring is
    optional
(END)
```

Functions: Scope

Local Scope

Variables that are defined inside of functions.

Global Scope

Variables that are defined at the global level.

Built-in Scope

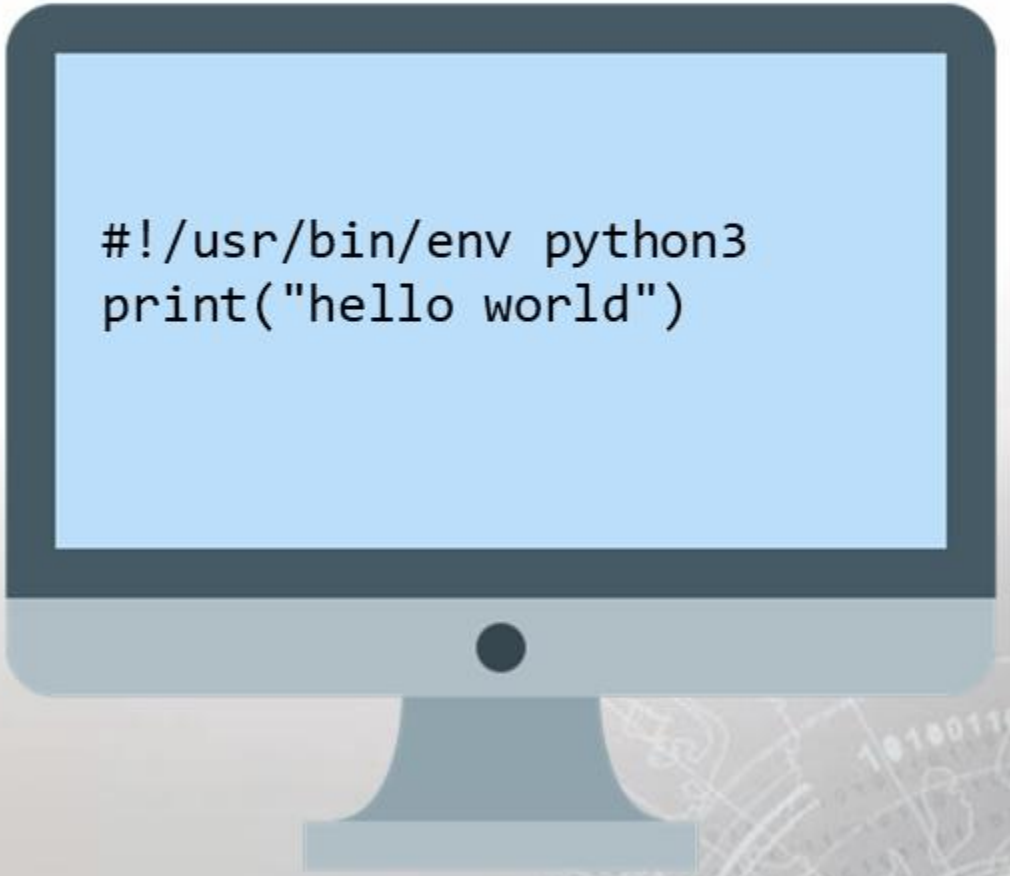
Variables that are predefined in Python.

```
>>> x = 2 # Global
>>> def scope_demo():
. . .     y = 4 # Local to scope_demo
. . .     print("Local: {}".format(y))
. . .     print("Global: {}".format(x))
. . .     print("Built-in: {}".format(dir))
>>> scope_demo()
Local: 4
Global: 2
Built-in: <built-in function dir>
```

Starting A Script

Which to use?

- `#!/usr/bin/env python3`
- `#!/bin/python3`
- `#!/usr/bin/python`
- `#!/usr/bin/pythonX.Y`
- `#!/usr/bin/python3.3`
- `#!/bin/bash`
- `#!/bin/sh`



Exercise: Python Refresher II

Objectives

After completing this exercise, students will be able to:

- Create conditionals, loops, and functions in Python
- Manipulate conditionals, loops, and functions in Python

Duration

This exercise will take approximately **2** hours to complete.

Debrief: Python Refresher II – Conditionals & Iterations

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

Debrief: Python Refresher II – Conditionals & Iterations

Specific Questions

- How many numbers does this code print?

```
i = 42
while i>=0:
    print(i)
    i=i-1
```

- What does `i` equal after the following code is ran?

```
i = 0
while i < 6:
    print(i)
    i += 1
```

Debrief: Python Refresher II – Conditionals & Iterations

Specific Questions

- What is the result of the following code?

```
i=0
while i < 6:
    print(i)
```

- How many numbers does the following code produce?

```
i = 10
while True:
    print(i)
    i=i-2
    if i <=2:
        break
```

Debrief: Python Refresher II – Conditionals & Iterations

Specific Questions

- What is the output of the following code?

```
concourse = [4,3,2]
concourse[1] = concourse[2]-1
if 1 in concourse:
    print(concourse[0])
else:
    print(concourse[1])
```

Debrief: Python Refresher II – Conditionals & Iterations

Specific Questions

- What is the output of the following code?

```
names = ['eric', 'karat', 'kathy']
if 'kathy' in names:
    print(names[1])
```

Debrief: Python Refresher II – Conditionals & Iterations

Specific Questions

- What is the output of the following code?

```
names = ['woody','travis','matt']
names.insert(1,'jordan')
if 'jordan' == names[1]:
      print(names[2])
```

Debrief: Python Refresher II – Functions

General Questions

- How did you feel about this procedure?
- Were there any areas in particular where you had difficulty?
- Do you understand how this relates to the work you will be doing?

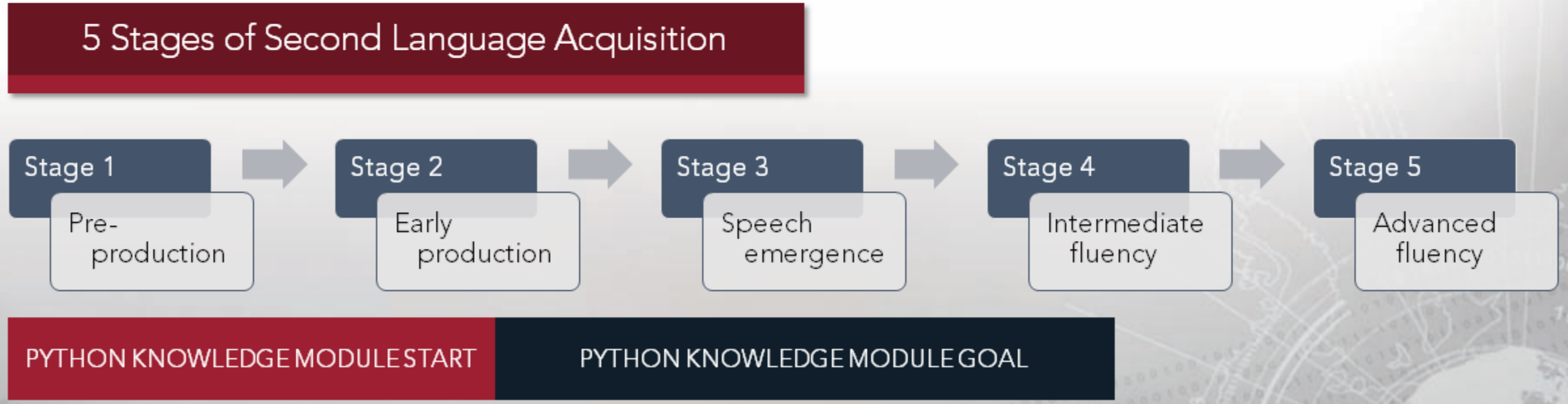
Specific Questions

- What are the key elements to creating a function?

Debrief

Python is a language… and learning a new language can be intimidating

- Continue to solidify your knowledge
- Engage your classmates and help one another



Resources

For additional information on and practice with using Python, explore the following suggested resources:

- "Illustrated Guide to Python 3" by Matt Harrison
- Michael Kennedy https://blog.michaelckennedy.net/category/python/
- "Effective Python Penetration Testing" by Rejah Rehim
- "Violent Python" by TJ. O'Connor
- Contributors to https://www.w3schools.com
- "Python Tricks The Book" by Dan Bader
- https://python.org

Lesson Summary

In this lesson we learned how to:

- Manipulate variables, strings, lists, dictionaries, conditionals, loops, and functions in Python
- Create variables, strings, lists, dictionaries, conditionals, loops, and functions in Python

End of Module 2, Lesson 9