TEKsystems
Own change

Intensive Introduction to Python

Participant Guide

7437 Race Road Hanover, MD 21076

COURSE CODE IN1467 / 12.18.2024

This course covers *Intensive Introduction to Python*

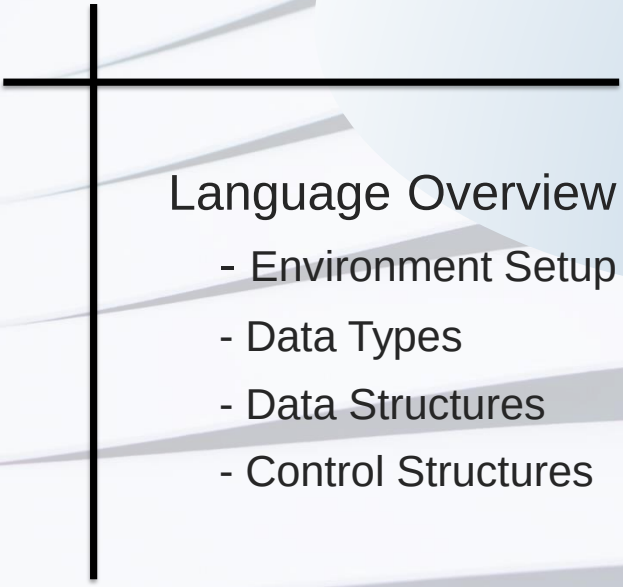
This version updated: 12/18/2024.

Notes

Chapters at a Glance

Notes

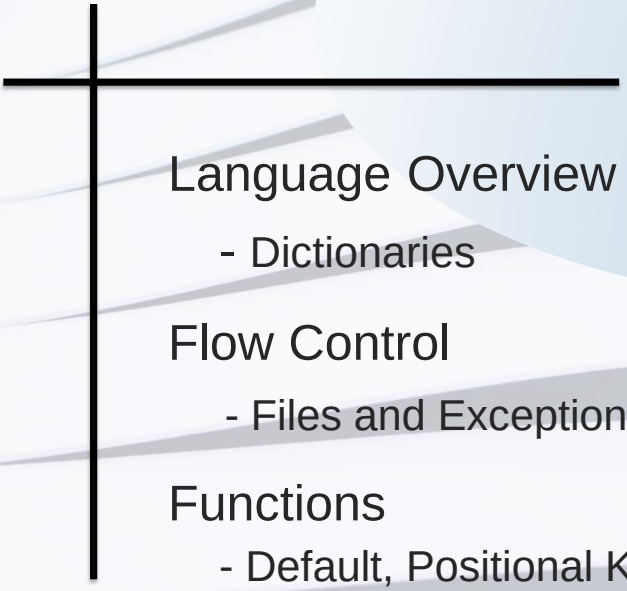
Course Agenda - Day 1



Language Overview

- Environment Setup
- Data Types
- Data Structures
- Control Structures

Course Agenda - Day 2



Language Overview *(continued)*

- Dictionaries

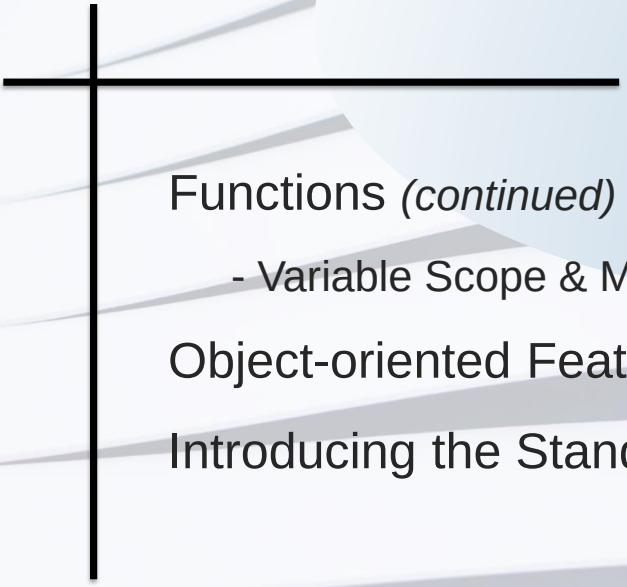
Flow Control

- Files and Exception Handling

Functions

- Default, Positional Keyword Arguments

Course Agenda - Day 3



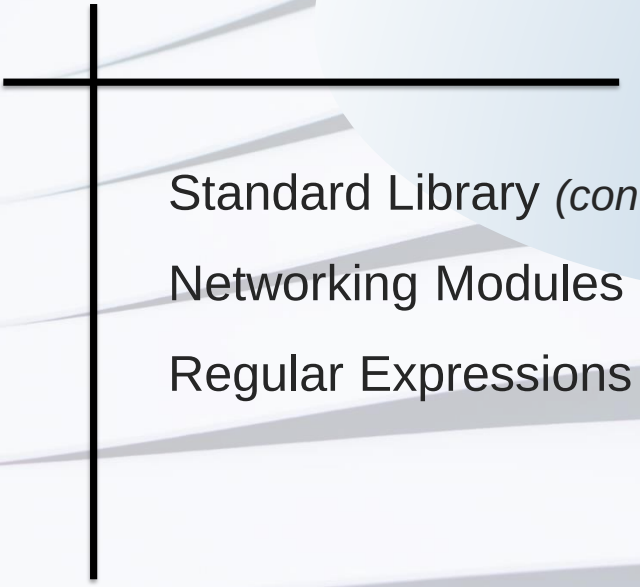
Functions *(continued)*

- Variable Scope & Modules

Object-oriented Features

Introducing the Standard Library

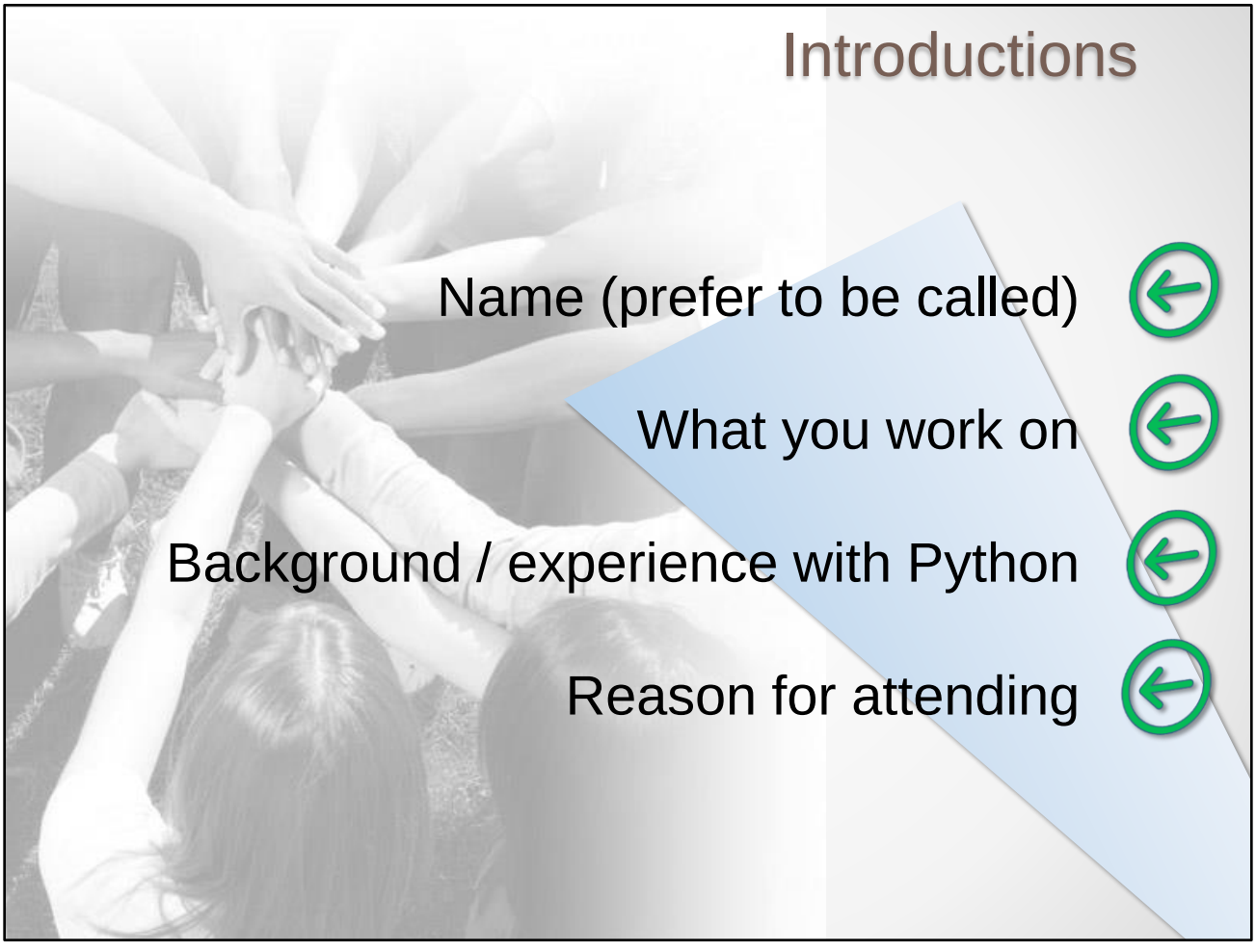
Course Agenda - Day 4



Standard Library *(continued)*

Networking Modules

Regular Expressions



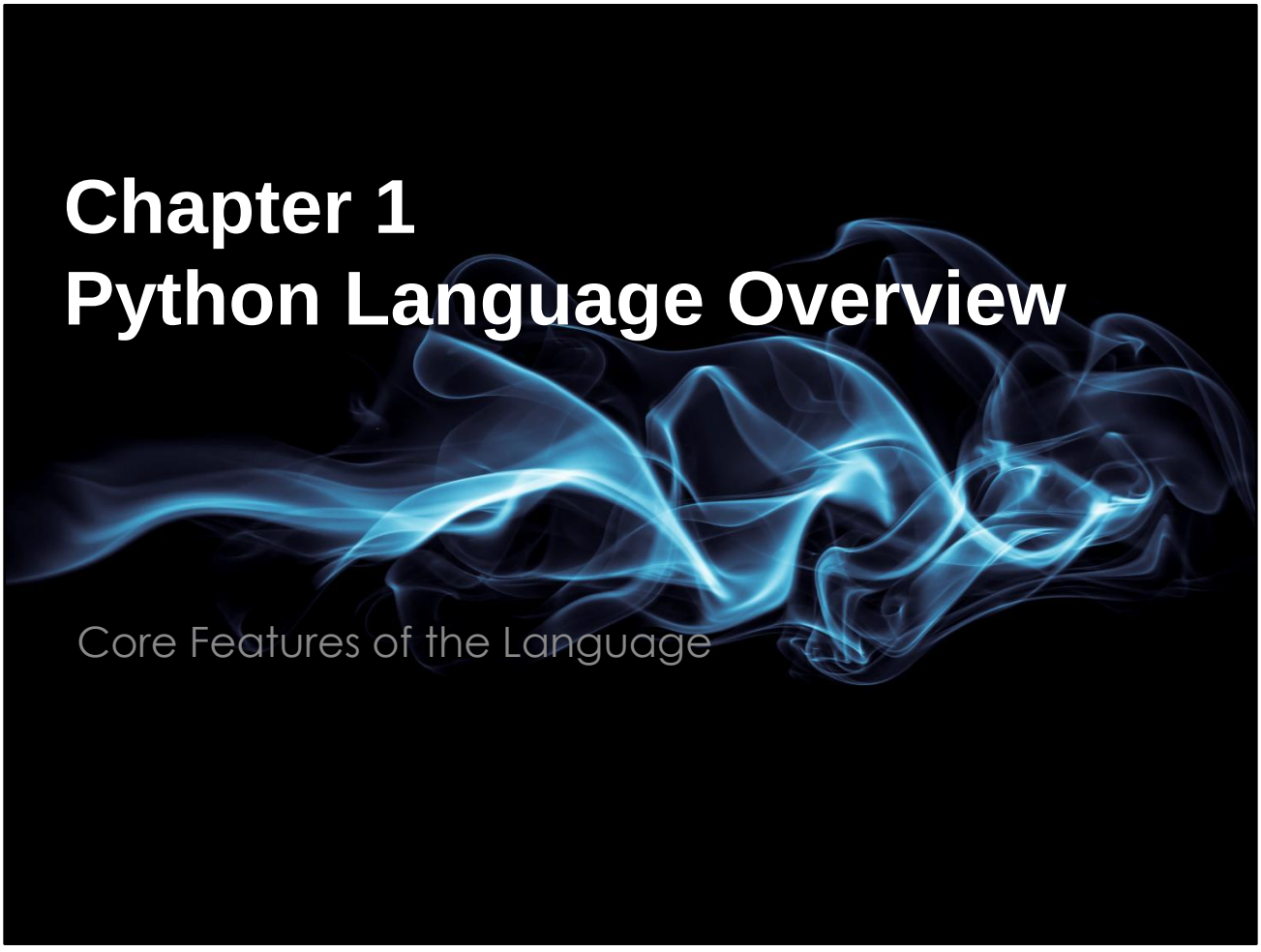
Typical Daily Schedule*

9:00	Start Day
10:10	Morning Break 1
11:20	Morning Break 2
12:30 – 1:30	Lunch
2:40	Afternoon Break 1
3:50	Afternoon Break 2
5:00	End of Day

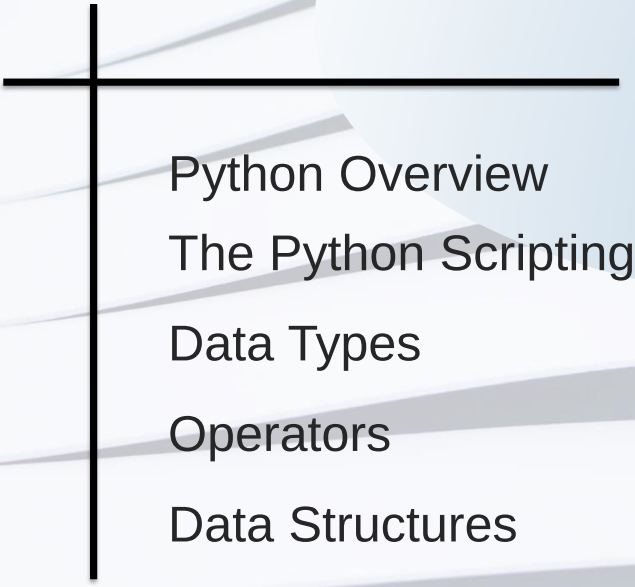
** Your schedule may vary, timing is approximate*

Get the Most from Your Experience

Ask Questions



Chapter 1 - Overview



Python Overview

The Python Scripting Environment

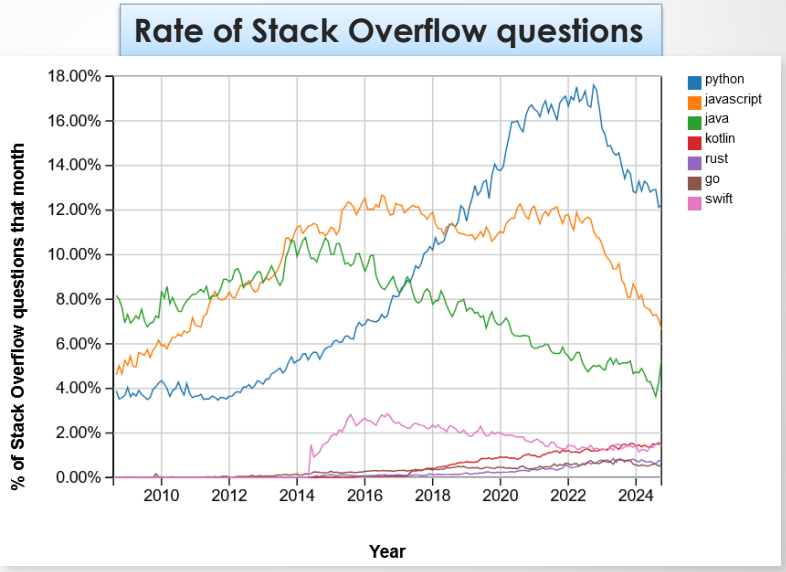
Data Types

Operators

Data Structures

Introducing Python

- *High-level,* general-purpose programming language
- Supports *imperative*, *object-oriented*, and *functional* programming styles
- Dynamic typing, automatic memory management



Python is a very expressive language. It incorporates numerous programming styles including imperative, object-oriented, and functional programming, yet it is not particular to any one of them.

The chart shows the amount of interest (by percentage of questions asked) in each language: Python, Java, and JavaScript and others over the last 14 years. (*https://insights.stackoverflow.com/trends*)

Language Origins

- Creator: *Guido van Rossum*
  - Netherlands
  - Began work on it in the late 1980s
  - Benevolent Dictator for Life (BDFL)
  - Worked at Google, Dropbox, most recently Microsoft
- First Released in *1991*
  - Influenced by the *ABC* programming language
- Named after *Monty Python's Flying Circus*

Python arrived over three decades ago. During this time, it has evolved, matured, and improved with each new release. While it still has idiosyncrasies, the language is used for numerous purposes in modern application development.

The Python creator, Guido van Rossum, was a fan of Monty Python and wanted a quirky name for the language (*https://www.linkedin.com/in/guido-van-rossum-4a0756*).

Why Python in the Enterprise?

- It's free, open source, and easy to read
- It runs cross-platform (mostly) and integrates with languages such as C/C++

- The two most powerful features of Python
  - Powerful Python Standard Library
  - Large, active, contributing developer community

- Most popular uses are for
  - Systems and testing automation
  - Data analytics and sciences
  - Web application development (incl. API Development)

Two import aspects of Python include the standard library (numerous modules that ship with Python and improve its capabilities) and an active third-party developer community. Contributions from thousands of other developers allows Python to solve programming problems that can often prove to be much more work in other languages.

Why NOT Python in the Enterprise?
To a lesser degree than in earlier years, Python code (which runs within an interpreter) can execute slower than natively compiled languages such as C++ and even Java. Therefore, if performance is a primary concern, Python may not be able to compete with some languages. Performance, however, has been a major focus in recent versions. Python 3.13 is faster than any prior version, for example.

Version History

- Python version numbers follow the format

major.minor.patch
(example: 3.13.1)

Which version am I using? Open a command/terminal window. Type:
python -V or
python3 -V or
python3.13 -V (where 13 represents the minor version, try 3.12, 3.11, 3.10, etc.)

- Notable versions

Python <3.9 is end-of-life!

Version	Date	Comments
1.0	1991	Initial release
2.0	2000	
2.4	2004	Decorators
2.5	2006	*with* statement, try/except/finally combo
2.6	2008	print() as a function, string formatting methods
2.7	2010	Final Python 2.x minor release
2.7.18	2020	Python 2.x fork end-of-life
3.0	2008	Not 2.x compatible, avoid using 3.0-3.2
3.6	2016	f-strings, NamedTuple, improved dict perf.
3.7	2018	Dataclasses, typing module enhancements
3.10	2021	match-case control
3.11	2022	Better exception messages, typed dicts
3.13	2024 (Oct)	Free threads

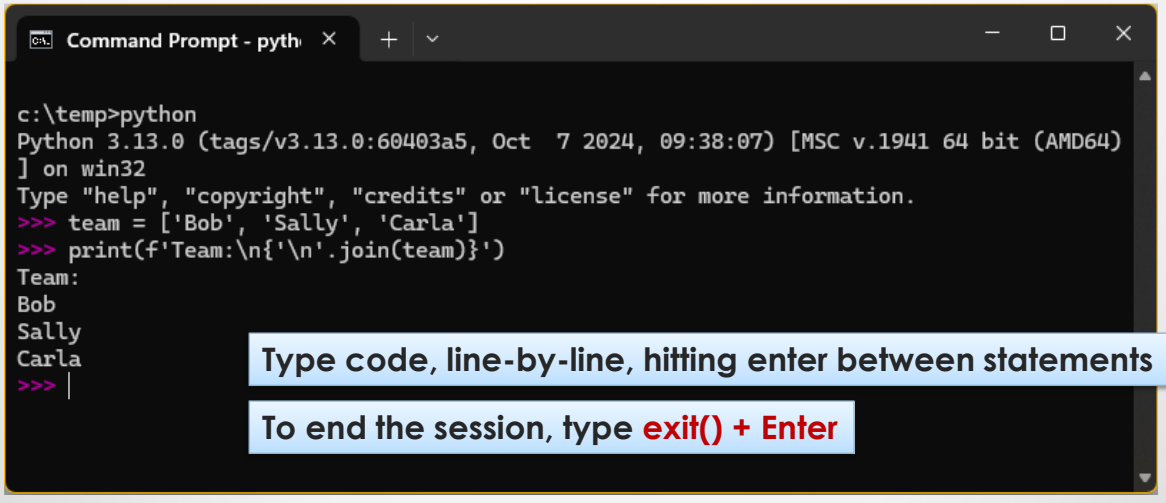
Breaking changes occurred going from Python 2 to 3. Python 2.7.18 was the last 2.x release, and no more maintenance releases will occur on Python 2 (as of Jan. 2020). Legacy Python 2 code should be ported to Python 3. As of this writing, Python 3.8 and earlier have been end-of-lifed (no further changes pushed to them).

In addition to Python versions, there are also Python "flavors". These include:
CPython - the typical default Python. The interpreter is written in C and is the one most used when executing Python scripts.
IronPython - Written in C# and runs within the .NET VM. C# classes can be imported as well as limited Python Standard Libraries.
Jython - compiles Python code into Java ByteCode. Uses JDK classes and Python libraries but compiles into .class files and runs within a JVM.
Other versions exist: ActivePython (by ActiveState), Stackless (supporting micro-threads), winpython (scientific windows portable version), PyObjC (Objective-C version of Python), Brython (JavaScript), RubyPython and more.

Executing Code within a Shell

- Python code can be executed line-by-line within the Python *shell* (console)

Launch the shell by typing python, or python3, or python3.13, or python3.x *(where x matches your Python minor version)*



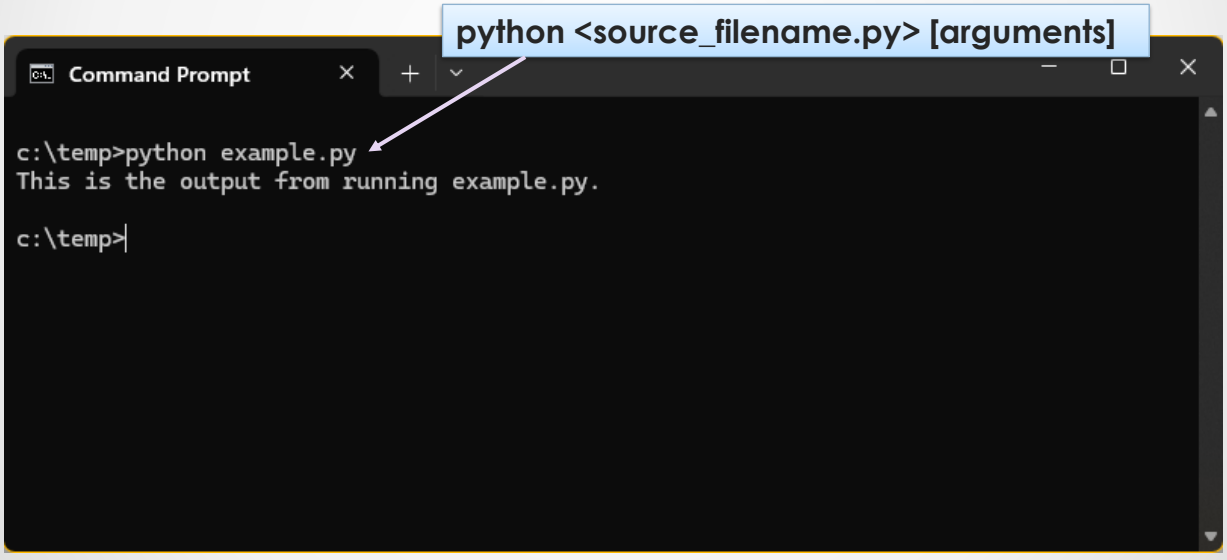
To open and run a script from within the Python 3 shell (not common to do), type exec(open('<path>/<filename>.py').read()).

Note: on Windows OS, typing a specific version, e.g., python3.13, doesn't work out of the box.

Note: the code example shown above only works in Python 3.12+ (due to the use of nested single-quotes).

Executing Code within Source Files

- Typically, code is written in a text file that ends with .py and then executed using the Python interpreter



On Unix and Linux systems, a "shebang" line (e.g., #!/usr/bin/env python) can be included in the source file that defines how to execute the script.

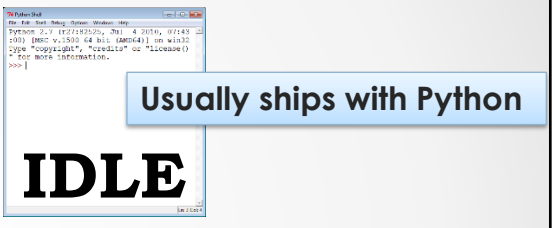
#!/usr/bin/env python3

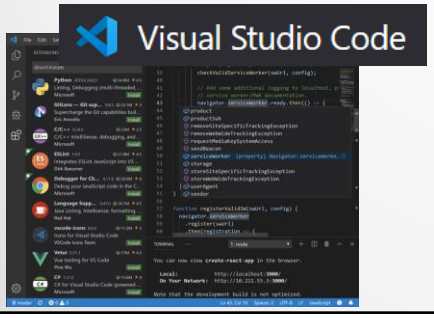
Via a shebang line within a source file

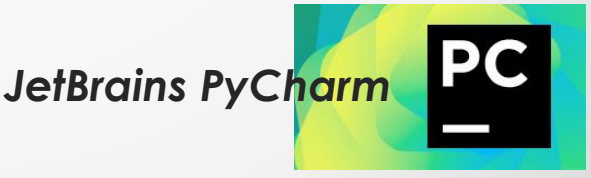
Scripts can also be executed from within development environments, such as IDEs like PyCharm.

Python IDEs

- Numerous options exist for writing Python code



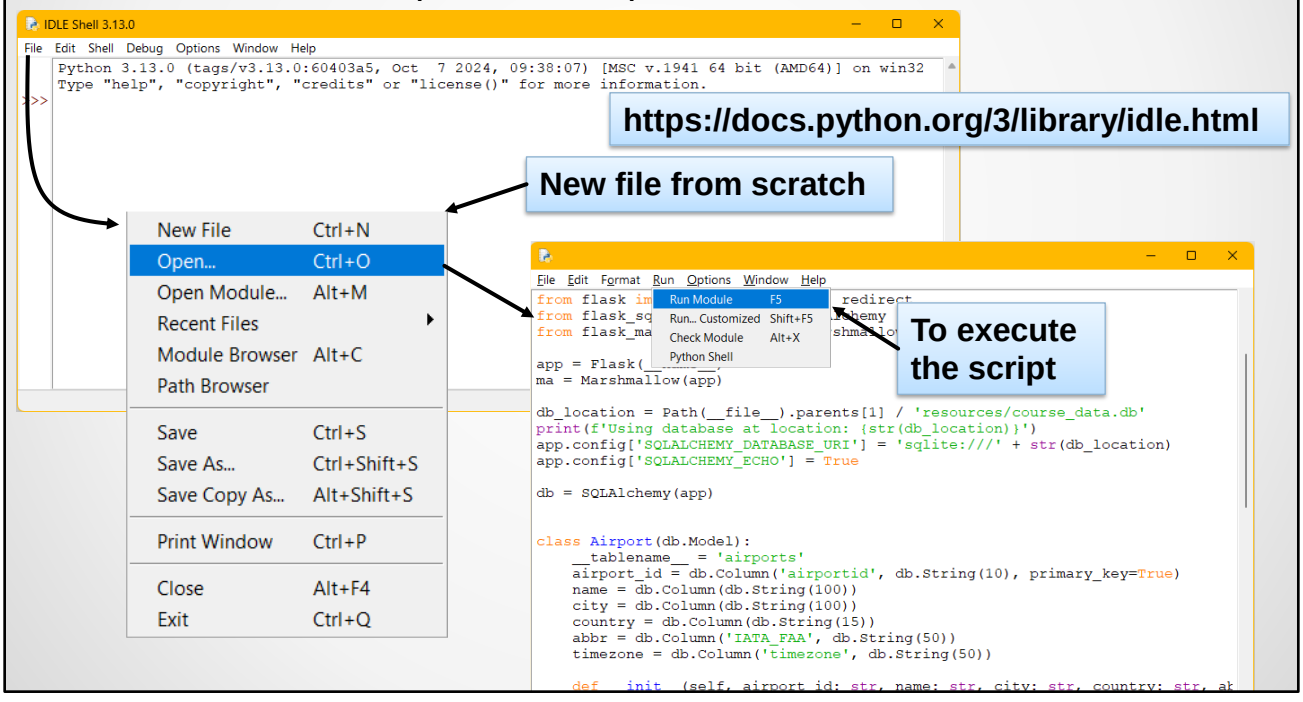




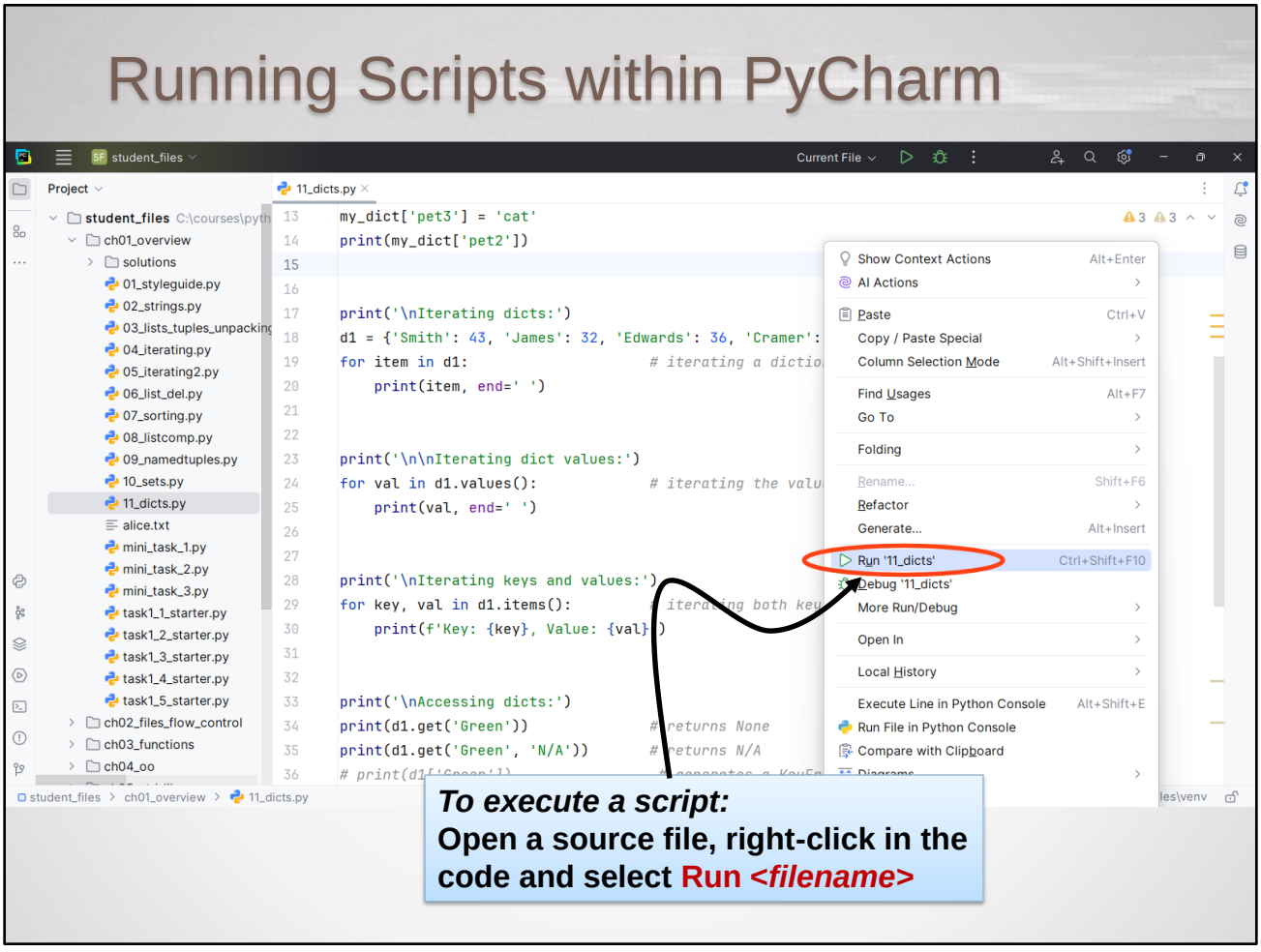
All of these environments are either free or have a free version including JetBrains PyCharm Community Edition.

Using IDLE

- IDLE is an editor that ships with all Python versions
  - Useful for simple, fast script creation



Idle can be found in the <PYTHONHOME>/Lib/idlelib directory. Launch it in one of several ways: either from an icon in the Start menu (Windows), or by typing its name (idle) from the command-line, or by running **python -m idlelib** (where ***python*** should be python, python3, python3.13, etc., depending on your environment).



PyCharm is both an advanced IDE and one of the most popular and powerful options that exists. It is an easy-to-use editor supporting numerous programming languages and file formats as well as a tool that provides support for many specific Python frameworks such as Pandas, NumPy, Matplotlib, Django, Jupyter, Sphinx, and so many more.

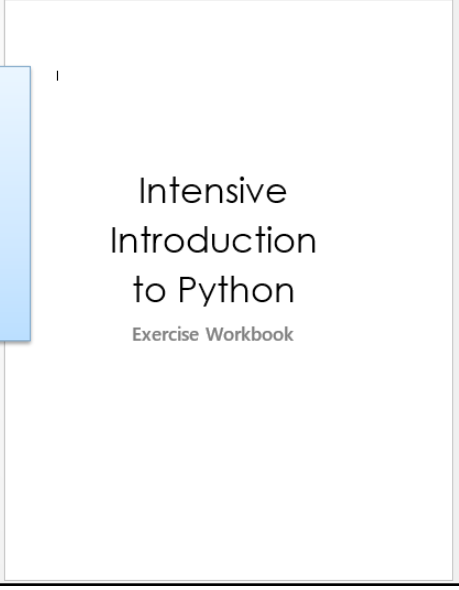
Usually, a good first step is to verify the correct interpreter is selected for use. Do this by visiting File > Settings > Project: student_files > Project Interpreter and examining the currently selected interpreter.

Your Turn! Task 1-1

Python Environment Setup and Test

- Install Python, test the interactive shell, and configure the student files within the IDE

Locate the instructions for this exercise in the back of the student manual



Task 1-1 Questions

Try to answer these questions based on the behavior of the application:

1. *task1_1_starter.py* refers to sys.argv. What does sys.argv do?

2. What does sys.argv[1] represent?

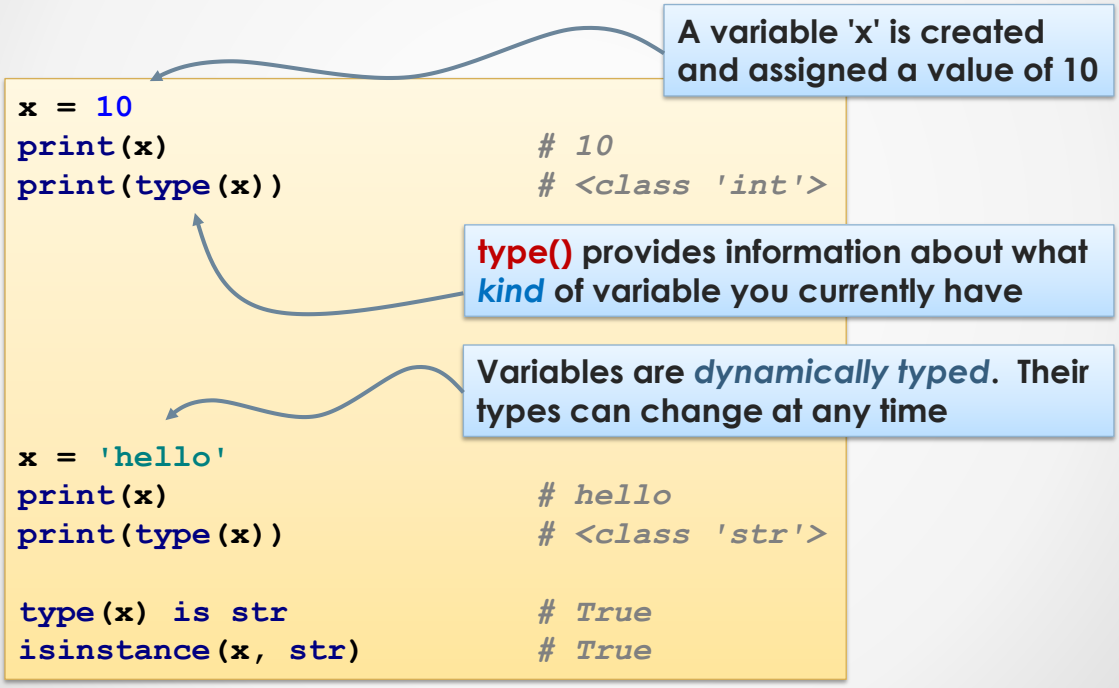
3. Based on this, what is sys.argv[0]?

Answers:

1. Collects the filename and user-defined arguments provided from the command-line.
2. The first argument provided after the name of the script.
3. The name of the script.

Python Data Types

- Python has many built-in data types



Unlike strictly typed languages, like Java, when a variable is created, it is not given a type. The type is determined when the variable is assigned a value. This doesn't happen until runtime. Because variables aren't given types until runtime, this affects the entire way in which we program with this language. Our patterns of usage will be very different than those of other programming languages; and therefore, we shouldn't bring those programming practices with us to this language.

Core Built-in Data Types

Value	Description	
int	integer	
float	floating point	Numbers
complex	complex numbers	
bool	booleans	
str	strings	
bytes	fixed array of bytes	
bytearray	array of bytes	Sequences
list	ordered item sequence (arrays)	
tuple	fixed ordered item sequence (fixed arrays)	
dict	unordered collection of key/value pairs (hashes)	
set	unordered collection of unique items	
object	objects	
function	functions	
file	File objects	

Python's complex type is not common. The bytes and bytearray types to deal with the fact that str types are now represented using Unicode characters.

In Python, there is not an actual sequence type, this is an abstract concept used to group together some related types that share several operations and capabilities. We'll discuss those capabilities shortly.

Python Style: PEP 8

- Python identifiers are case sensitive

```
greet, Greet, GREET = 'hello', 'howdy', 'hola'
```

- Python defines hundreds of special documents, called PEPs (Python Enhancement Proposals)
- PEP 8, one of the most common, defines the recommended style for writing Python code
  - It is not a requirement to follow this guideline, but it is recommended

https://www.python.org/dev/peps/pep-0008/

student_files/ch01_*overview/01_styleguide.py*

Variable naming conventions are similar to those encountered in other languages.

The PEP 8 document is short and can be read during lunchtime.

Some PEP 8 Conventions

- Use 4 spaces for indentations

```
if datetime.now().isoweekday() == 5:
    print("TGIF!")
```

4 spaces, no tabs

- Never mix spaces and tabs. Prefer spaces over tabs.
- Other important PEP 8 rules
  - Two blank lines between top-level functions and classes
  - One blank line between methods in classes
  - Use blank lines in functions to group logical sections
  - Put imports at the top of a file
  - Put imports on separate lines

```
import os, sys
```

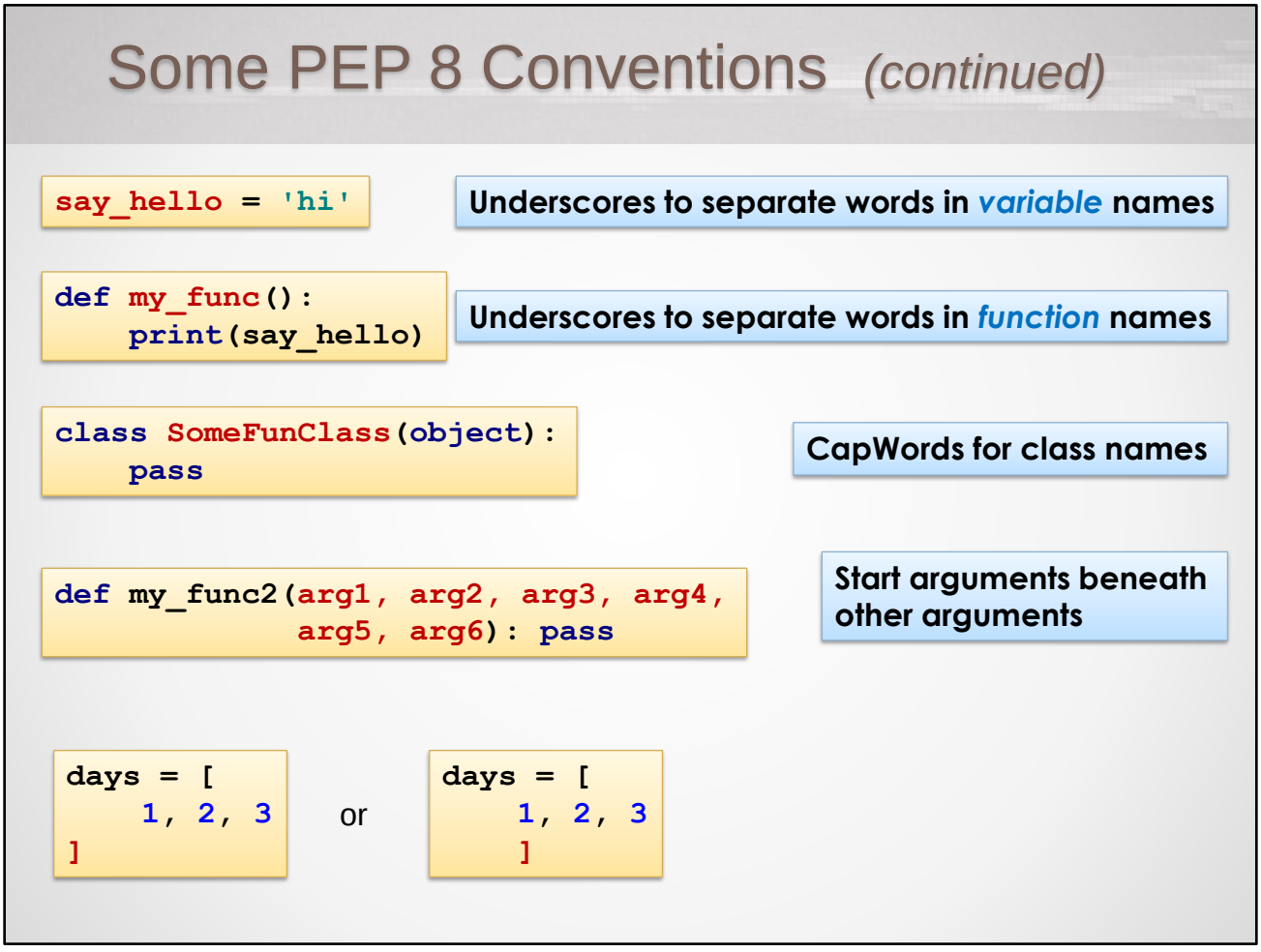
```
import os
import sys
```

student_files/ch01_overview/01_styleguide.py

It is okay to import multiple items from a single module on the same line, otherwise this would be inefficient. Example:

```
from datetime import date, datetime
```

Importing multiple items this way would be okay because they are from the same module.



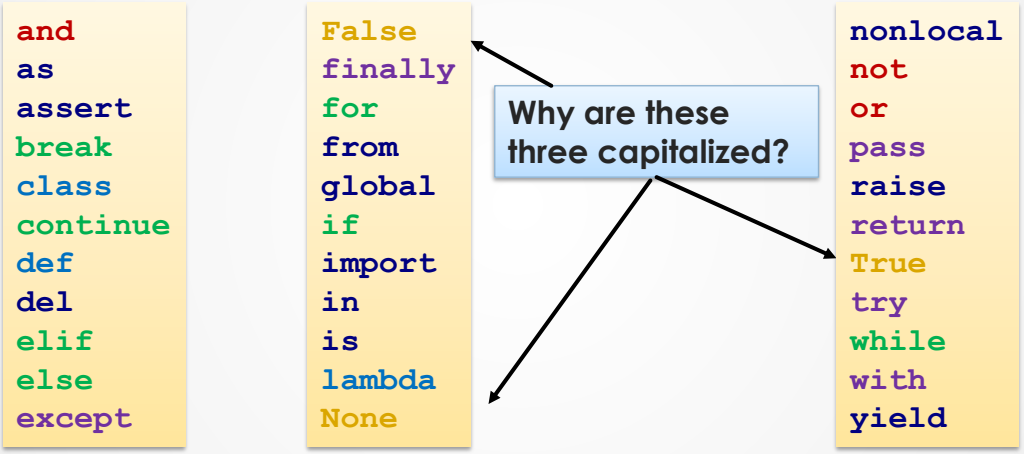
student_files/ch01_overview/01_styleguide.py

While these rules are defined in the PEP 8, they are only suggestions. However, most developers follow these naming conventions. The main reason for the PEP 8 guide is to promote readability across Python packages and projects.

The *pass* keyword (shown above) is formally introduced later. However, for now we'll mention that it is used as a placeholder for an empty block of code in Python.

Python Keywords

- Don't use these reserved words as identifiers



- Don't use Python's built-in types, classes, functions, exceptions, etc. as variable names
  - For example: open, len, list, etc...

In addition to the keywords listed above, you should also not create identifiers from anything defined in the builtins module.

Perform an `import builtins` followed by `dir(builtins)` to see this additional list.

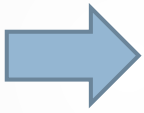
Strings

- Strings are *immutable sequences of Unicode characters*
  - Type: str
  - Formal creation:
  - Literal creation:

```
my_str = str('Python is great!')
```

```
my_str = 'Python is great!'
```

```
my_str = 'Python is fun'
```



```
my_str = 'Python is very fun'
```

A new string object gets created in memory

- Strings support both *single* and *double* quotes

```
my_str = 'Python\'s great fun'
```

```
my_str = "Python's great fun"
```

```
my_str = """Python's
    great
fun"""
```

PEP 8 does not specify a preference

Triple (double) quotes is a multi-line string with whitespace retained. A common use is for documenting code (docstring)

student_files/ch01_overview/02_strings.py

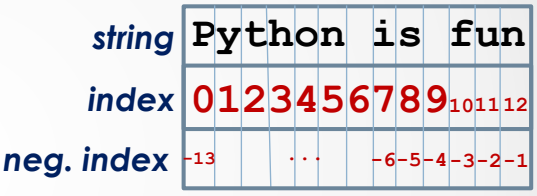
In Python 3, strings are Unicode characters. Some characters must be escaped to use them. Escape characters include:

\\	backslash
\'	single quote (as shown in the slide)
\"	double quote
\b	backspace character (ascii)
\n	linefeed
\r	carriage return
\t	tab

String Random Access and Slicing

- Strings are *sequences* of characters
  - They can be accessed (indexed) using square brackets

```
my_str = 'Python is fun'
print(my_str[0])        P
print(my_str[-13])      P
print(my_str[13])       IndexError
print(my_str[-14])      IndexError
```



- Sequences may be *sliced* (sub-sequenced)

```
new_string = string[start : end : step]
```

```
print(my_str[0:9])      Python is
print(my_str[:3])       Pyt
print(my_str[3:])       hon is fun
print(my_str[-3:])      fun
```

student_files/ch01_overview/02_strings.py

Two things to note about indexing and slicing:

1. Negative values supplied will wrap around from the end.
2. Slices return new objects in memory.

String Features and Methods

- String concatenation

```
my_str = 'Python is '
new_str = my_str + 'fun'
```

Concatenation creates a new string

- *string.*split(sep)

```
my_str = "Python is great"
my_list = my_str.split(' ')
```

Splits a string based on a specified separator character (default splits on whitespace)

['Python', 'is', 'great']

- *string.*replace(substring, new_string)

```
my_str.replace('is', 'is still')
```

Replaces all matching substrings with new_string

Python is still great

student_files/ch01_overview/02_strings.py

There are numerous string methods. These are just a few of the available methods and capabilities built into the string type.

The join() method can join several strings each separated by the specified character (string) as shown in the example below:

```
places = ['First', 'Second', 'Third']
print('_'.join(places))                # Result:  First_Second_Third
```

By default, split() will break a string up based on any whitespace characters and it will split it as many times as necessary. split() supports a second argument, maxsplit, that can limit the number of times a split occurs.

Additional String Methods

- *string*.strip()

Returns a new string with whitespace removed from each end

```
new_str = '    Whitespace will be removed.      '.strip()
```

- idx = *string*.find(substring)

Finds the index of the first occurrence of the substring

```
my_str = 'Python is great'
my_str.find('is')

my_str.find('not')
```

7

-1

- str(obj)

Converts anything into a string

```
s = str(55)

s = str(3.14)
```

'55'

'3.14'

student_files/ch01_overview/02_strings.py

The str() method is a special case that creates a new string object from any object passed to it.

Conversions

- Use the *type* name as a function to perform conversions

```
s1 = str(55)
s2 = str(3.14)

i1 = int('37')
i2 = int(3.14)

f1 = float(55)
f2 = float('3.14')
```

'55'
'3.14'

37
3

55.0
3.14

- If a conversion cannot be made, a ValueError is raised

```
int('hello')
```

ValueError: invalid literal for int() with base 10: 'hello'

What we described above isn't totally accurate. We are doing more than just calling a function when we use str(), int(), and float(). We are actually creating new objects. For example, int('37') creates a new Python integer object.

Using *string*.format()

- To structure output, use the *string*.format() method of the str class:

{index : format_spec}

```
s = 'It has been raining for {0:.2f} {1} and {0:.4f} {2}'


new_str = s.format(40.001, 'days', 'nights')
```

Position 0 Position 1 Position 2

'It has been raining for 40.00 days and 40.0010 nights'

- Partial syntax of *format_spec*:

```
:[ fill ][ align ][ width ][ group ][ .prec ]
```

Extra padding char	< (left) > (right)	Field width	Group char (e.g., 1,000)	Num digits after decimal place

student_files/ch01_overview/02_strings.py

The format() method was once very important to string formatting. However, since Python 3.6, its usage has sharply declined due in large part to the arrival of f-strings (discussed shortly). If you are using Python 3.6+, prefer the use of f-strings over format() whenever possible.

More on the format() method syntax can be found at:
https://docs.python.org/3/library/string.html#formatspec

format() with Keywords

- *string*.format() supports keyword arguments also

```
s = '{0} is over {age:0.2f} {time} old.'.format('Python',
                                                time='years',
                                                age=25)

print(s)
```

Python is over 25.00 years old.

- Additional Examples

```
'{0:10}{1:10}{age:>10}'.format('John', 'Smith', age=37)
```

Field widths

John Smith 37

10 10 10

```
print('{:-^20}'.format('hello'))
```

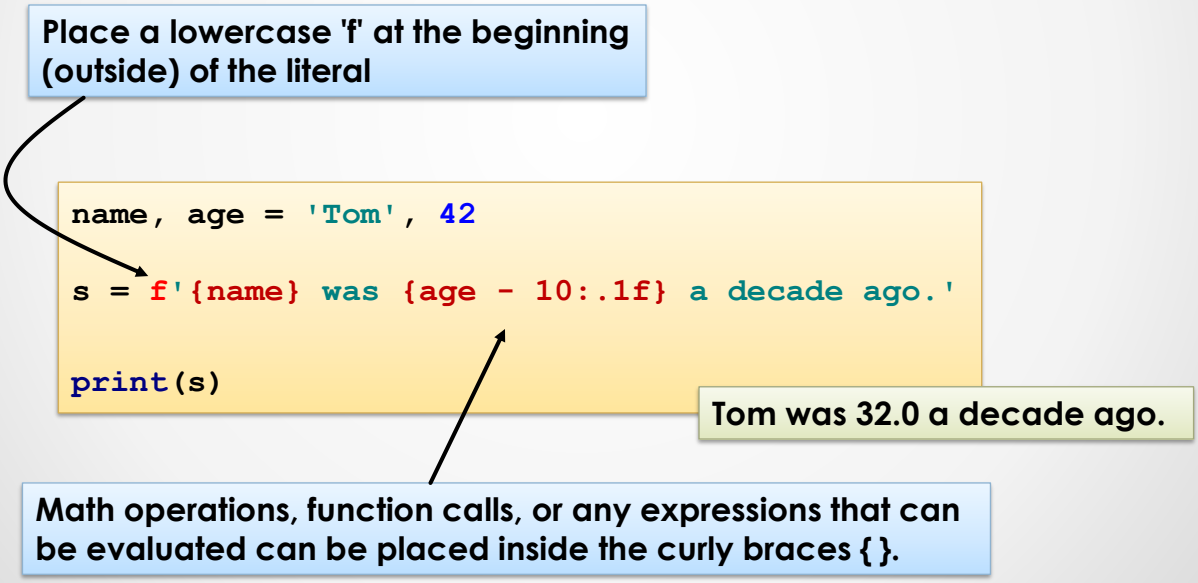
-------hello--------

Arguments provided in a function call, like time='years', are known as ***keyword arguments***.

Keyword arguments reduce the dependency on order within format() or within any function but must be indicated using an equals (=) operator. They must also appear *after* any positional parameters. More will be discussed about positional and keyword arguments in a later chapter.

f-strings (Formatting Strings)

- Python 3.6 introduced f-strings, or formatted strings
  - f-strings support variables that have been previously declared



You must be using Python 3.6 or later to take advantage of f-strings. Using f-strings in earlier versions of Python will cause exceptions to occur.

f-strings are faster than using format(), %, or + in string operations.

Other String Methods

- Some additional string methods

```
print(' '.join(['Tim', 'Sally', 'John']))
```

Tim Sally John

```
print(f'Team members:\n{'\n'.join(['Tim', 'Sally', 'John'])}')
```

Team members:
Tim
Sally
John

Python 3.12+, escape characters and nested similar quotes now supported in f-strings

capitalize()
casefold()
center(width, char)
encode(encoding)
endswith(str)
index(substr)
join(sequence)
lstrip()
lower()
rfind()
removeprefix()
removesuffix()
rindex()
rsplit()
rstrip()
splitlines(bool)
startswith(str)
upper()

casefold()	supports Unicode text and should be preferred over lower()
str.count(substr)	counts the number of occurrences of the substr
expandtabs()	returns new string with tabs replaced by 8 spaces
ljust(width, fillchar)	returns new string with the string left aligned in a string the size of width. fillchar can be a padding character.
splitlines(bool)	returns list of strings split on str, strips whitespace if bool is false (default)
endswith(substr), startswith(substr)	returns true or false if the substr ends or begins the string, substr may be a tuple of strings

Your Turn! Mini-Task 1

- *Extract the operating system and version from this string:*

```
ua = 'Mozilla/5.0 (Windows NT 10.0)'
```

- Work from the provided file: *mini_task_1.py*

- Use an f-string to display the results in the following way:

```
OS: Windows NT 10.0
```

Sequences

- Sequences (str, list, tuple) are ordered collections of objects that support common features

```
dirs = ['North', 'South', 'East', 'West']
```

- Random access/slicing

```
dirs[2]
```
East

```
dirs[-2:]
```
['East', 'West']

- Concatenation

```
dirs + ['NW', 'NE', 'SW', 'SE']
```
['North', 'South', 'East', 'West', 'NW', 'NE', 'SW', 'SE']

- Length (size)

```
len(dirs), len(dirs[0])
```
4, 5

- Membership

```
if 'East' in dirs:
    print(dirs.index('East'))
```
2

Sequences exhibit similar behaviors such as the ability to be randomly accessed, perform slicing, support membership checks, and have their size (length) returned through the **len()** function.

In addition, sequences also support a min() and max() function.
Examples of min() and max():

```
max([1973, 2001, 2015, 2013, 1994])
min([1973, 2001, 2015, 2013, 1994])
```

min() and max() support a key parameter that can be supplied as an argument to define what biggest or smallest means. The key parameter is discussed in the section on dictionaries later.

Lists

- Lists are *mutable*, ordered collections of objects
  - Creating lists

```
my_list = []
my_list = list()

my_list = [1, 3, 5]
my_list = [3.3, 'hello', 'hello', 3.3]
my_list = list('hello')
```

Empty lists, first one preferred

Lists may contain duplicates

Converts another sequence type to a list

  - Adding objects to lists

```
my_list = [1, 2, 3]
my_list.append(10)
my_list.insert(1, 'hello')
```

1, 'hello', 2, 3, 10

student_files/ch01_overview/03_lists_tuples_unpacking.py

Lists may be created using literal notation with square brackets [ ]. While literal notation is most common and preferred, occasionally lists are created using the list() type name.

Use append() to add items onto the end of the list.

To add a value into the middle of a list, use insert(position, value), keeping in mind that sequences are zero-based in Python.

Tuples

- Tuples are *immutable*, ordered collections of objects

```
my_tuple = ()
my_tuple = tuple()

my_tuple = (1, 3, 5)
my_tuple = (3.3, 'hello', 3.3)
my_tuple = tuple('hello')
my_tuple = (1,)
```

Empty tuples, first one preferred

Converts another sequence to a tuple

Special case, one-element tuple

- Modifying a tuple causes a TypeError

```
contact = ('John Smith', ['123 Main St', 'Los Angeles', 'CA'])
contact[0] = 'Jonathan Smith'
```

TypeError: 'tuple' object does not support item assignment

```
contact[1][0] = '456 Elm St'
```

This is allowed, but why?

student_files/ch01_overview/03_lists_tuples_unpacking.py

Tuples do not have append(), insert(), or other methods that attempt to modify the data as this would violate the concept of a tuple.

Some like to describe the difference between tuples and lists as:
- tuples have structure while lists have order, or
- tuples are heterogeneous (hold different kinds of things) while lists are homogeneous (hold same kind of things)

Example: a database in Python commonly returns a list of tuples. The list contains same type records while the tuple contains different fields.

Your Turn! Mini-Task 2

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sinotech.com')
]
```

- The above data structure can be found in the ch01_overview/mini_task_2.py file
- *Edit and run this script* to perform the following
  - Display Sally Edwards age. Can it be changed?
  - Add a new record into *records*
  - Display how many records you then have after this
  - Display how many fields are in the second record from the end (using negative indexing)
  - Display how long Keith Cramer's email is

Invalid Operations

- Can't concatenate different sequence types

```
[1, 2, 3] + 'Bob'
[1, 2, 3] + (4, 5, 6)
```

'+' must be used on similar type sequences (e.g., list + list)

- Can't add things into a list if the position doesn't exist

```
my_list = []
my_list[0] = 'Bob'
```

Either use append, or create a list with the items in it and then change the value

```
my_list = ['Sam', 'Abe', 'Eva']
my_list[0] = 'Bob'
```

Sequence Unpacking

- Unpacking in Python allows for individual variables to be assigned to fields in a sequence

```
person = ('Carla', 'Esposito', 44, 'cesp@bywaysmast.com')

first, last, age, email = person
print(person[1], last)
```

Esposito Esposito

```
first, last, *other = person
print(other)
```

[44, 'cesp@bywaysmast.com']

```
first, last, age, email, *other = person
print(other)
```

[ ]

```
first, *other, email = person
print(other)
```

['Esposito', 44]

student_files/ch01_overview/03_lists_tuples_unpacking.py

A commonly used feature of Python is the unpacking feature. This allows values within sequences to be assigned to individual variables for easy access.

The "star" notation implies "collect." It stores into a list the remaining items not represented by individual variables.

Control Structures

- Python's primary form of branching is found in the if - elif - else control

There may be zero or more elif branches

The else is optional

```
if test:
    <one or more statements>
elif test2:
    <one or more statements>
else:
    <one or more statements>
```

Beginning and ending of blocks are determined by indentations (use 4 spaces)

```
if test:
    print('If test is true, this will execute')
    print('So will this!')
print('This will always execute!')
```

Acceptable values for *test* are discussed on the next slide.

Python 3.10 introduced a new branching structure, the match-case control which represents the first time Python has ever had a switch type control structure. This is mentioned a little later.

Truthy / Falsey Values

- The *test* evaluates to either True or False

Any expression that can be evaluated can be placed in the test condition (e.g., numbers, strings, objects, function calls, math operations, etc.)

- *test* evaluates to True except for the following values

False	''
None	[]
0	()
0.0	{}

```
data = [1, 2, 3]
if len(data):
    print('Data not empty.')
```

3

Data not empty.

- Python logical operators *short-circuit*

```
data = None
...
if len(data):
    print('Data not empty')
```

TypeError possible

```
data = None
...
if data and len(data):
    print('Data not empty')
```

Guards against None

The empty curly braces (in the orange box above) represent a dictionary (discussed later). Empty data structures are evaluated as *False*. { } are dictionaries and/or sets in this case. In either case empty dictionaries and sets evaluate to False. [ ] represents an empty list, which also evaluates to False. ( ) represents a tuple. Empty tuples are also False.

In the middle example above, the length of the list is evaluated to 3 which isn't any of the "Falsey" conditions, therefore, the if condition evaluates to True!

In the bottom example, the left side risks *data* remaining *None*. Any operations on *data* (including a len() check) will cause an error if *data* is *None*. Therefore, it is best to check if *data* is *not None* first. The right side illustrates this. The **and** operator will short-circuit, meaning the second condition after the **and** will not even be evaluated if the first condition evaluates to *False*. Note: this does not guard against the condition that data is not a list. Data could still be a value such as 10 and this code will fail.

To allow an *empty* condition to pass the if-test but not a *None* condition, the example in the lower right should be changed to:

```
if data is not None and len(data):
```

```
            print('Data not None.')
```

Comparison Operators

- Some of the common Python comparison operators

```
a == b
a <  b
a >  b
a >= b
a <= b
a != b
a is b          (same object)
a is not b
a in y          (member of)
a not in y
not a
a and b         (short circuits)
a or  b         (short circuits)
```

```
list1 = [1, 2, 3]
list2 = list1
list3 = list1[:]

print(list1 == list2)
print(list1 is list2)
print(list1 == list3)
print(list1 is list3)
```

Makes a copy

True

True

True

False

The ***is*** operator checks memory locations (meaning same object), the == operator looks at members to perform item by item comparison. list1 and list3 above are different objects but have the same members so == will be *True* while ***is*** yields a *False* value.

Iterative Control Structures

```
while test:
     <one or more statements>
else:
     <one or more statements>
```

else block is rarely used (it only executes if the loop terminates naturally without a break)

```
for one_item in iterable:
    <one or more statements>
else:
    <one or more statements>
```

The *for-loop* performs better than the *while-loop* and should generally be preferred

Examples Using the *for* Loop

```
week = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

for day in week:
    if day != 'Sun' and day != 'Sat':
        print('Weekday: ' + day)
```

```
Weekday: Mon
Weekday: Tue
Weekday: Wed
Weekday: Thu
Weekday: Fri
```

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]
for record in records:
    print(f'{record[0]} {record[1]}, {record[2]} {record[3]}')
```

```
John Smith, 43 jsbrony@yahoo.com
Ellen James, 32 jamestel@google.com
Sally Edwards, 36 steclone@yahoo.com
Keith Cramer, 29 kcramer@sintech.com
```

student_files/ch01_overview/04_iterating.py

Python's *for* and *while* loops are the only iterative control structures. There is not a classic 3 part for( ) loop as in C++, Java, and other languages. Both *while* and *for* support the *break* and *continue* statements.

Additional Ways to Iterate

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]
```

```
for record in records:
    print('{0} {1}, {2} {3}'.format(*record))
```

Expand record

```
for (first, last, age, email) in records:
    print(f'{first} {last}, {age} {email}')
```

Unpack record

Usually leave off parentheses ()

```
for first, last, age, email in records:
    print('{first} {last}, {age} {email}'.format(first=first,
    last=last, age=age, email=email))
```

student_files/ch01_overview/04_iterating.py

Each of these approaches yields the same output as the one on the previous slide.

Your Turn! Mini-Task 3

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sinotech.com')
]
```

- Work from ch01_overview/mini_task_3.py file
- Task: *Given a person's name (e.g., "Sally Edwards"), display their age*
- Hints:
  - Use split() to break the string into two parts
  - Compare the given first and last name to each first and last name in each record

Advanced: Using the rare else-block of the for-loop, display a message if the person's name is not found.

Your Turn! Task 1-2

- Finds the *host (domain) name* of a URL
  - Work with these three URLs

https://docs.python.org/3/
https://www.google.com/search?q=python
http://localhost:8005/contact/501

Our approach:

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

First strip this off

Then strip this off

Advanced
Make it to work for all 3 URLs iteratively

Choose your comfort level:

Experienced with Python: *Attack this task on your own, your way*

Experienced programmer, but not with Python: *Work from the additional hints in the source code (ch01_overview/task1_2_starter.py)*

Newer to Python and programming: *Step-by-step instructions are provided in the workbook in the back of the manual*

Since URLs come in a variety of formats, for this exercise, we will limit URLs to the ones shown above.

Tools for Task 1-3

- To open a file and read lines from it

Note: File handling is formally introduced in chapter 2.

```
for line in open(filename):
    # process one line from file
```

- To get a list of items in a directory

```
import os
os.listdir(directory_name)
```

- To check if a directory item is a file

```
if os.path.isfile(filename):
    # must be a file
```

Your Turn! Task 1-3

grep = get regular expression, a utility to locate the occurrence of expressions within files

- Create a simple *grep* utility
  - Syntax: python task1_3_starter.py wordexpression directory
  - Reads files from a directory, reads all the text files and returns the line number if the expression is found
  - Sample execution:

python task1_3_starter.py print .
File: iterating.py, Line: 8, (print)
File: iterating.py, Line: 20, (print)
File: listcomp.py, Line: 3, (print)
File: strings.py, Line: 3, (print)

Opens all files in the current directory and looks for the occurrence of the word "print"

This exercise only works with text, not binary, files

Pick your preferred level:
Experienced: solve this your own way
Some programming: use hints in source file
Novice: follow steps in back of manual

Reading files will be discussed in more detail later. For now, to read a line from a file, use open(filename) as follows:

```
for line in open(filename):
    # process one line from file
```

Overall, your steps should be something like this:

- Iterate over the file_list
- Open each file (from the list) one at a time
- Read each line in, check for the presence of the wordexpression
  Hint: Use .find() for this
- Output a result if there is a match

Additional Iterating Techniques

- Python provides multiple ways to help iterate

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]
```

```
for idx, contact in enumerate(records):
    print(idx, contact[1])
```

```
0 Smith
1 James
2 Edwards
3 Cramer
```

```
for contact in reversed(records):
    print(contact[1])
```

```
Cramer
Edwards
James
Smith
```

```
for idx, contact in enumerate(reversed(records)):
    print(idx, contact[1])
```

```
0 Cramer
1 Edwards
2 James
3 Smith
```

student_files/ch01_overview/05_iterating2.py

These handy, globally available functions assist in the iterating process. enumerate() provides a counting variable to keep track of the iteration count. It also accepts a second argument representing the starting value (default is zero).

Note: enumerate() takes any iterable while reversed() takes any sequence.

What's the difference? Sequences are all iterable, but not all iterables are sequences.

Also, it is worth noting that reversed(enumerate(iterable)) is not legal.

Another useful utility is the zip() function:

```
fruit = ['Apple', 'Orange', 'Banana', 'Watermelon']
color = ['red', 'orange', 'yellow', 'green', 'blue']

for f, c in zip(fruit, color):
    print(f'The {f} is {c}')
```

Other List Methods

index(item)	finds the first occurrence or -1	Find/ Count
count(obj)	number of occurrences of obj in list	
sort()	sorts in-place	Modifying
extend(lst2)	similar to list1 += list2	
reverse()	reverses the list order in-place	
copy()	shallow copy of list	
clear()	removes all items from the list	Removing
pop()	returns/removes last item	
remove(item)	removes first occurrence of item	

student_files/ch01_overview/06_list_del.py

The **del** operator may also be used to remove items from lists.

```
data = None
a = [1, 'foo', (), 3.3, str(data), data]
print(a)                                # [1, 'foo', (), 3.3, 'None', None]

# remove 2nd item
del a[1]
print(a)                                # [1, (), 3.3, 'None', None]

# remove first and second
# items after last removal
del a[0:2]
print(a)                                # [3.3, 'None', None]
```

Sorting

- Use sort() for simple, in-place sorting:

```
items = [37, 2, 0, -14]
items.sort()
print(items)
```

Lists only

[-14, 0, 2, 37]

- Use sorted() to return a *new list*
  - *sorted()* is useful for *non-lists* such as tuples
  - Use it when you don't want to modify the original

```
items = (37, 2, 0, -14)
new_list = sorted(items)

print(new_list)
```

Works on any Iterable, returns a list

[-14, 0, 2, 37]

student_files/ch01_overview/07_sorting.py

When determining whether to use sort() or sorted() consider your needs. If a new list is desired keeping the original intact, then use sorted(). However, if the added overhead of creating a new list doesn't make sense, when simply modifying the one in memory will suffice, then use sort(). Performance-wise, both are about the same with sort() perhaps having a slight advantage.

Sorting in Reverse

- Sorting in reverse will sort from largest to smallest

Sorts items in-place from largest to smallest

```
items = [37, -14, 0, 2]
items.sort(reverse=True)
print(items)
```

[37, 2, 0, -14]

Sorts items from largest to smallest but *returns a new list*

```
items = [37, -14, 0, 2]
new_list = sorted(items, reverse=True)
print(items, new_list)
```

[37, 2, 0, -14]

student_files/ch01_overview/07_sorting.py

Use *reverse=True* to reverse the sort order. Normally it will sort smallest to largest. *reverse=True* will cause it to sort largest to smallest.

Note: *reversed()* is not the same as *reverse=True*. It accepts a sequence and iterates it from the end to the start.

Sorting Using a Key

- Perform custom sorts using the key parameter
  - The return value identifies how to compare elements

```
nums = ['13', '1', '11', '4']
nums.sort()
print(nums)
```

Probably not what we wanted!

['1', '11', '13', '4']

Using key=

```
def sort_func(val):
    return int(val)

nums.sort(key=sort_func)
print(nums)
```

['1', '4', '11', '13']

```
nums2 = sorted(nums, key=sort_func)
print(nums2)
```

['1', '4', '11', '13']

student_files/ch01_overview/07_sorting.py

Sometimes the default sort is not ideal. In this example, we have a list of numeric strings. The default sort() sorts it using hexadecimal character values, which results in an ascii type sort not a numerical sort.

To fix this, we can provide a means for sorting. Using the **key=** keyword parameter, we can sort using a function that returns the value that should be used to compare each item in the list.

Introducing Lambdas

- Python provides functions to promote reusability

```
def calc_square(val):
    return val * val

calc_square(10)
```

- Lambdas are one-line functions that are written *inline*
- *Syntax*

lambda inputs : ret_val

```
calc_square = lambda val: val*val

calc_square(10)
```

```
def sort_func(val):
    return int(val)
```

...equivalent to...

```
lambda val : int(val)
```

student_files/ch01_overview/07_sorting.py

In general, use lambdas sparingly, but use them if the use case presents itself.

Lambda limitations:

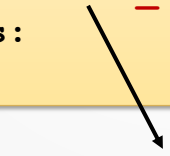
- The expression must be a single expression, not a whole statement (no equals sign). Basically, what you are allowed to have in a return statement.
- No variable assignments are allowed and no if or for loops are allowed

Functions are formally introduced in chapter three.

Sorting Using a Key *(continued)*

- Sorting a list of records by age (descending)

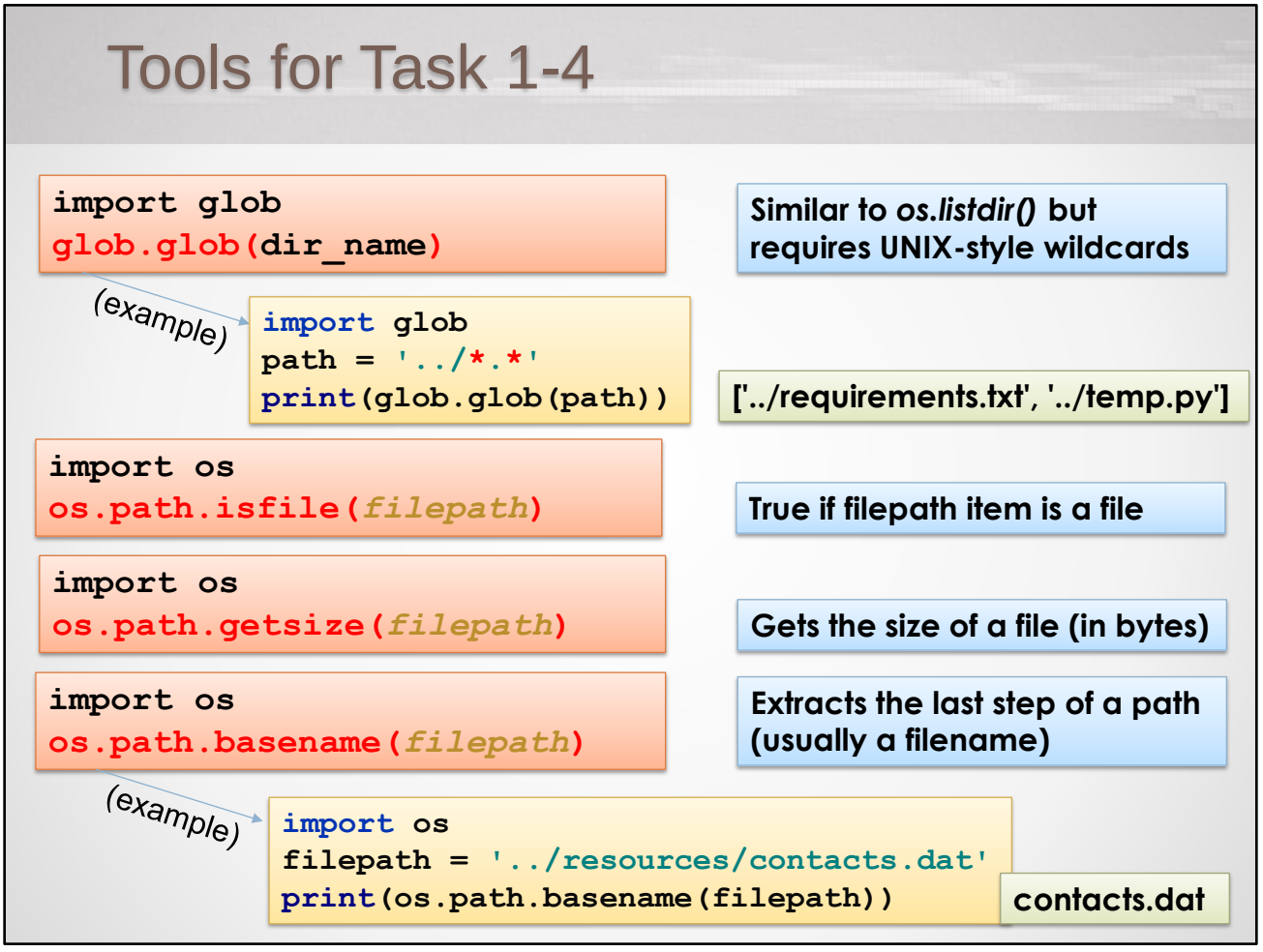
```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]

records.sort(key=lambda one_rec: one_rec[2], reverse=True)
for record in records:
    print(record)
```



```
[('John', 'Smith', 43, 'jsbrony@yahoo.com'),
 ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
 ('Ellen', 'James', 32, 'jamestel@google.com'),
 ('Keith', 'Cramer', 29, 'kcramer@sintech.com')]
```

student_files/ch01_overview/07_sorting.py

This example sorts a list of tuples. It sorts records in descending order of their age (reverse=True) by using the key parameter. The lambda receives one tuple at a time, upon which it returns one_rec[2] or the age field. This means that it will sort by the ages.



glob() from the glob module can be used to retrieve items in a directory. Under the hood, glob.glob() uses os.listdir().

Other uses:

```
glob.glob('*')      # matches all directory contents
glob.glob('*.py')   # matches all .py files
```

Your Turn! Task 1-4

- Create a script that sorts and displays files in a directory by size *(largest first)*
- Directory contents are already obtained, merely remove subdirectories and sort the files
- Format results to include the file size on the output

Sample output:

```
alice.txt                          167518
task1_3_starter.py                   2841
task1_2_starter.py                   2024
task1_5_starter.py                   1915
02_strings.py                        1832
07_sorting.py                        1584
08_listcomp.py                       1363
11_dicts.py                          1111
task1_4_starter.py                   1054
01_styleguide.py                      897
03_lists_tuples_unpacking.py          849
04_iterating.py                       772
task1_1_starter.py                    654
10_sets.py                            633
05_iterating2.py                      611
mini_task_2.py                        534
09_namedtuples.py                     476
06_list_del.py                        381
```

Follow additional instructions within ch01_overview/task1_4_starter.py

List Comprehensions

- List comprehensions provide a Pythonic way to make lists from other lists (or iterables)
  - Like *for-loops*, they take the following form

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

```
list1 = [1, 2, 3, 4, 5]
list2 = [x*2 for x in list1]
print(list2)
```

[2, 4, 6, 8, 10]

```
new_list = [expression for var in iterable test_condition]
```

```
list1 = [1, 2, 3, 4, 5]
list2 = [x*2 for x in list1 if x % 2 == 0]
print(list2)
```

[4, 8]

Only considers even numbers

student_files/ch01_overview/08_listcomp.py

The expanded version of the second example above would look like this:

```
list1 = [1, 2, 3, 4, 5]
for x in list1:
     if x % 2 == 0:
         list2.append(x*2)
print(list2)
```

Task 1-4 Revisited *(Using List Comprehensions)*

- The following uses a list comprehension to complete the previous task (Task 1-4)

```
dir_contents = []
path = '.'
match = '*'
for pathitem in glob.glob('/'.join([path, match])):
    dir_contents.append(pathitem)

files = [(os.path.basename(item), os.path.getsize(item))
         for item in dir_contents if os.path.isfile(item)]

files.sort(key=lambda fileinfo: fileinfo[1], reverse=True)

for name, size in files:
    print(f'{name:<20}{size}')
```

student_files/ch01_overview/08_listcomp.py

Introducing Named Tuples

- Named tuples are (like) tuples that support using dot operators in addition to indices
  - Simpler than creating classes
  - They are ordered, easier to read than dictionaries

- Consider the earlier example:

```
records = [
    ('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    ('Ellen', 'James',   32, 'jamestel@google.com'),
    ('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    ('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]

records.sort(key=lambda one_rec: one_rec[2], reverse=True)
print(records)
```

student_files/ch01_overview/09_namedtuples.py

This is a reworking of the sorting sample that we saw earlier. Next, we'll convert it to a named tuple...

Using Named Tuples

```
from collections import namedtuple

Contact = namedtuple('Contact', 'first last age email')

records = [
    Contact('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    Contact('Ellen', 'James',   32, 'jamestel@google.com'),
    Contact('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    Contact('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]

records.sort(key=lambda one_rec: one_rec.age, reverse=True)


for record in records:
    print(record.last, record.age)
```

Import the namedtuple() type from the collections module

Class type

The type name

Can be a *space-separated string* of properties or a *list of strings*

student_files/ch01_overview/09_namedtuples.py

The syntax is to declare a namedtuple() followed by the name of the type and a space-separated string (or list of strings) that identifies the properties of the new named tuple. The return type from the namedtuple() function is a newly created class type. You may then create records using the new class type as shown above.

Notice that each record will now have properties that match the names you provided when creating the type.

Also, named tuples are ordered, meaning subscript notation still works (a[0] and a.first are equivalent in this example).

Advantages over regular tuples: they are still accessed like tuples, but they can additionally support property-style access. Advantages over dictionaries: they are ordered and have a better readability than the ugly syntax of dictionaries. Advantages over classes: they are simpler to create and work with!

Sets

- Sets are collections that disallow duplicate items
  - Unordered, mutable, iterable
  - Support len(), in, comparisons
  - Members must be hashable (immutable) such as int, float, str, tuple
  - Disallows list, dict, set, custom objects
  - **frozenset** is an *immutable* set version

```
s1 = set([1, 2, 3, 2])
print(len(s1))
```

3

Set literal notation

```
s2 = {4, 5, 6}
print(s1.isdisjoint(s2))
```

True

isdisjoint() - no elements in common

```
s3 = frozenset([2, 4, 7])
print(s2.difference(s3))
```

(5, 6)

difference() - elements in s2 not in s3

student_files/ch01_overview/10_sets.py

The set constructor takes an iterable. Only hashable (meaning immutable) items can be added to a set. Attempting to add a non-hashable item will result in a TypeError. Use **clear()** to empty a set. **remove()** can remove individual elements. If the element is not in the set() a KeyError will be raised. **isdisjoint()** returns *True* if two sets have no elements in common. **difference()** returns a set of the elements in the caller set that do not appear in the argument set. copy() makes a shallow copy of a set.

The example below uses records defined as a set and disallows the 4th record because it is a duplicate:

```
records = set()
records.add(('John', 'Smith', 43, 'jsbrony@yahoo.com'))
records.add(('Ellen', 'James', 32, 'jamestel@google.com'))
records.add(('Sally', 'Edwards', 36, 'steclone@yahoo.com'))
records.add(('Ellen', 'James', 32, 'jamestel@google.com'))    # not added, duplicate
print(len(records))                                           # 3
```

Dictionaries

Other languages may refer to these as *hashes*, *hashmaps*, *maps*, *lookups*, *tables*, etc.

- Dictionaries are collections of name/value pairs
  - Ordered (as of Python 3.6), mutable, iterable
  - Support len(), in, comparison operators
  - Keys are hashable (immutable), values can be anything

```
d = { key1: value1, key2: value2, ... }
```

```
my_dict = {}
my_dict = dict()
my_dict = { 'pet1': 'dog', 'pet2': 'fish' }
my_dict = dict(pet1='dog', pet2='fish')


my_dict['pet3'] = 'cat'
print(my_dict['pet2'])
```

empty dicts

Literal notation

No quotes on keys here

adding / changing values

accessing values

Dictionaries are useful data structures as they provide a means to store any kind of data yet access that data via a key. Because dictionary keys must be unique, attempting to add an entry to a dictionary that has the same key in existence will replace the value with the new value.

Accessing Dictionaries

```
d1 = {'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29}
```

- Direct access can generate a KeyError

```
d1['Smith']
d1['Green']
```

```
43
Generates a KeyError
```

- Use exception handling to deal with a KeyError

```
try:
    value = d1['Green']
except KeyError:
    value = 0
```

```
Generates a KeyError
```

- dict.get(key) to retrieve values also:

```
d1.get('Edwards')
d1.get('Green')
```

```
36
None
```

- dict.get(key, default) is safest

```
d1.get('Cramer', 0)
d1.get('Green', 0)
```

```
29
0
```

student_files/ch01_overview/11_dicts.py

Several ways to work with and access dictionaries are shown above. The safest of these techniques uses the get() method and supplies the desired key along with a default (fallback) value.

Iterating Over Dictionaries

```
d1 = {'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29}
```

- Iterating a dictionary directly returns *keys*

```
for item in d1:
    print(item)
```

```
Smith
James
Edwards
Cramer
```

- Iterating a dictionary's values

```
for val in d1.values():
    print(val)
```

```
43
32
36
29
```

- Iterating and using both key and value simultaneously

```
for key, val in d1.items():
    print(f'Key: {key}, Value: {val}')
```

```
Key: Smith, Value: 43
Key: James, Value: 32
Key: Edwards, Value: 36
Key: Cramer, Value: 29
```

student_files/ch01_overview/11_dicts.py

The above example illustrates that a dictionary specified within a `for`-control always returns the keys.

The items() method returns a "view" object instead of a copy of a list of tuples.

Note about dictionary views:
dict.items(), dict.keys(), and dict.values() all return "views" that are iterable. A view is a read-only iterable, however if the dict the view refers to is changed, the view is changed also.

Sorting Dictionaries

```
d1 = {'Smith': 43, 'James': 32, 'Edwards': 36, 'Cramer': 29}
```

- *dicts* are ordered and can be sorted by keys or values

Passing the entire dict into sorted() sorts and returns keys automatically

```
sorted(d1)
sorted(d1.values())
```

['Cramer', 'Edwards', 'James', 'Smith']

[29, 32, 36, 43]

- Entire dict sorted on *values*

```
list3 = sorted(d1.items(), key=lambda item: item[1])
```

Gives back a list of tuples

[('Cramer', 29), ('James', 32), ('Edwards', 36), ('Smith', 43)]

Convert back to dict type

```
back_to_dict = dict(list3)
print(back_to_dict)
```

{ 'Cramer': 29, 'James': 32, 'Edwards': 36, 'Smith': 43 }

student_files/ch01_overview/11_dicts.py

In the lower example, list3 is created by passing d1.items() which returns a list of tuples. The key= parameter tells sorted() to sort on the values, not the keys.

In the final code example at the bottom, the resulting sorted list of tuples above is passed into the dict() type function and it is converted back to a dictionary, now sorted by values.

Your Turn! Task 1-5

- Read the **ch01_overview/alice.txt** text file, and count the number of word occurrences into a dict
- Print the top 100 words (that are five characters or more) that occur the most

```
for line in open(filename, encoding='utf-8'):
    words = line.split()
```

words will be a list of strings

- **Read all words from the file**
- **Add each word into a dictionary**
- **On repeated words, increment its count**

To find the top occurring words, sort the dictionary by its values.

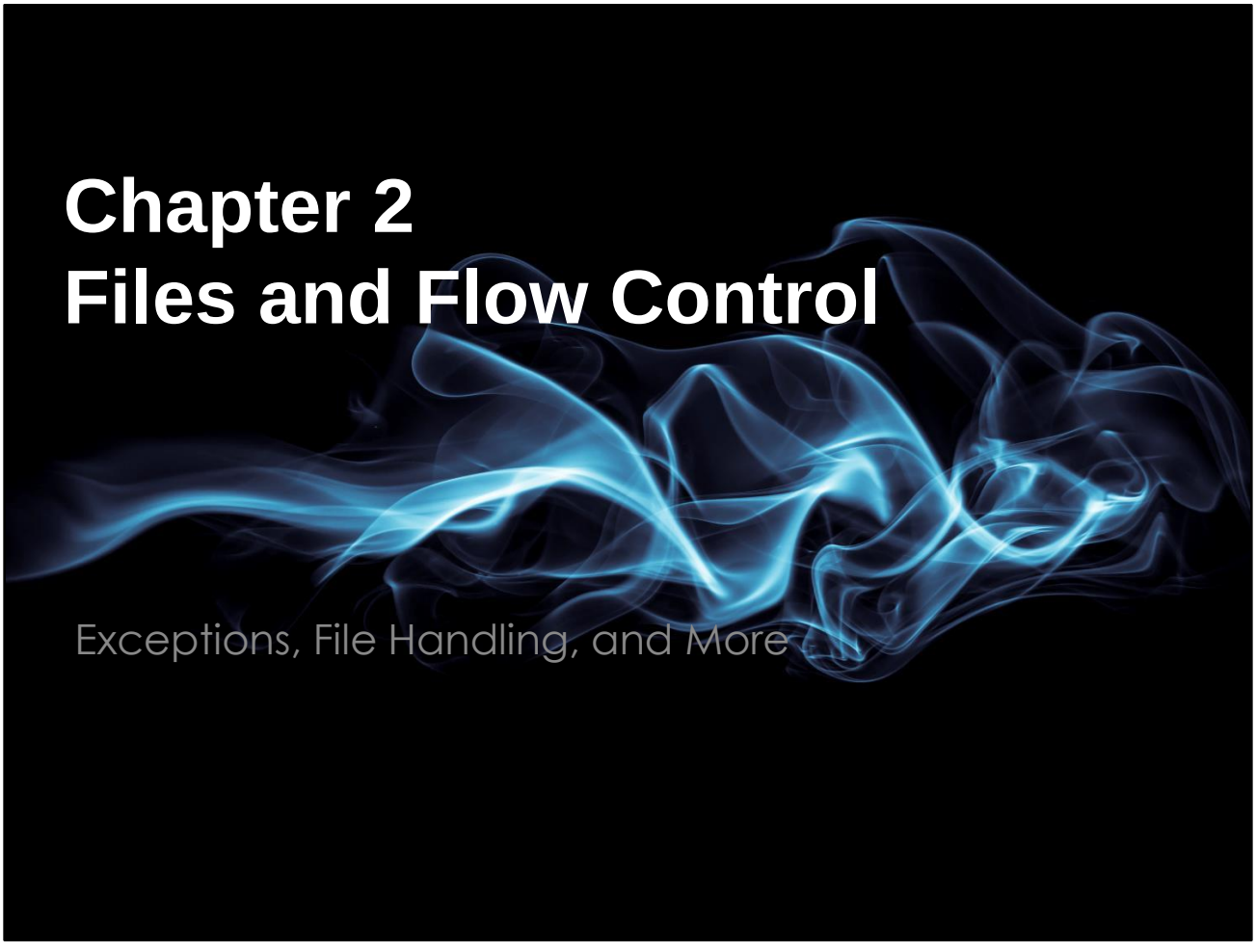
Follow additional instructions within ch01_overview/task1_5_starter.py

For this task, we aren't using regular expressions. You can optionally add exception handling, and the use of the *with* control if you know how to handle these.
Hint: the following sorts a dictionary by its values returning a list of the following: [(key1, value1), (key2, value2), ...]

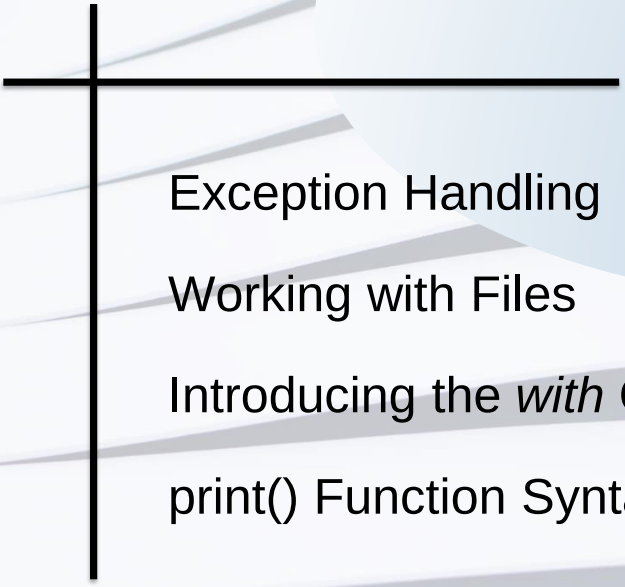
sorted(dictionary.items(), key=lambda a: a[1])

Chapter 1 - Summary

- Python is a high-level programming language with an active developer community
  - The language is popular for many reasons including its very easy-to-read syntax and its variety of uses
- A suite of data structures including lists, tuples, sets, dictionaries, and more help support Python's data manipulation capabilities
- Techniques for comparing, sorting, and managing data are provided via numerous built-in modules, classes, and functions



Chapter 2 - Overview



Exception Handling

Working with Files

Introducing the *with* Control

print() Function Syntax

Exception Examples

- The term "exception" indicates that something *exceptional* has occurred when executing your code

```
print('hello ' + 17)
```

```
Traceback (most recent call last):
  File "my_script.py", line 3, in <module>
    print('hello' + 17)
          ~~~~~~~~~^~~~
TypeError: can only concatenate str (not "int") to str
```

Python 3.11 introduced improved Traceback messages

```
print(1/0)
```

```
Traceback (most recent call last):
  File "my_script.py", line 3, in <module>
    print(1/0)
          ~^~
ZeroDivisionError: division by zero
```

```
(1, 2).append(3)
```

```
Traceback (most recent call last):
  File "my_script.py", line 3, in <module>
    (1, 2).append(3)
    ^^^^^^^^^^^^
AttributeError: 'tuple' object has no attribute 'append'
```

Exceptions in Python come in all flavors. The exception handling system in Python is optional unlike the checked exception system found in Java. Exception handling must be added in by the user.

Exception Handling Structure

- Exceptions are handled using the try - except - finally mechanism

At a minimum, either an except or finally is required

```
try:
    unsafe code
except exception1 as e1:
    exception1 handling
except exception2 as e2:
    exception2 handling
...
else:
    rarely used (called if
    try block is successful)
finally:
    always perform
```

optional

Alternate version without except

```
try:
    unsafe code
finally:
    always perform
```

Exceptions in Python are used to capture and respond to unexpected (exceptional) conditions. The flow would be as follows:

Under normal conditions:	try then else then finally
Under exceptional conditions:	try then except then finally

Handling Exceptions

```
a = input('Number 1: ')
b = input('Number 2: ')

try:
    a = float(a)
except ValueError:
    a = 0
try:
    b = float(b)
except ValueError:
    b = 0

result = a / b

print(f'Division result is: {result}')
```

Dangerous operation needs handling to prevent a crash

Improper values, such as *hello* generate a *ValueError*. Flow jumps to except block. The exceptions here have been handled individually

Yikes! This works but what happens here if 'hello' is entered for b?

student_files/ch02_files_flow_control/01_exceptions.py

When an exception occurs, flow jumps to the except block. If the potential for multiple errors can exist, multiple except blocks should be used. Try to use the most specific exception types as possible. List the more specific exception types first.

Adding and Combining *except* Blocks

```
a = input('Number 1: ')
b = input('Number 2: ')
result = 'undefined'
try:
    result = float(a) / float(b)
except ValueError:
    pass
except ZeroDivisionError:
    pass

print(f'Division result is: {result}')
```

The *pass* keyword is used to denote blocks where no code was needed. It is equivalent to { } in other languages

Zero entered for *b* generates a ZeroDivisionError which flows to the second except block

```
a = input('Number 1: ')
b = input('Number 2: ')
result = 'undefined'
try:
    result = float(a) / float(b)
except (ValueError, ZeroDivisionError):
    pass
print(f'Division result is: {result}')
```

When exception types will be handled similarly, they can be grouped

student_files/ch02_files_flow_control/01_exceptions.py

In the first example, exception handling was used to guard against potentially dangerous problems (number conversion and zero divisions). If an error occurs, the chosen value for result was 'undefined'. Since *result* was already 'undefined', no special handling in either of the except blocks was needed. An empty block condition of pass was provided since actually leaving it blank (empty) would not be allowed in Python.

In the second example, since both exception blocks are handled the same way, we combined the exceptions into a group. Now, if either exception occurs, the except block will be called.

Generic Handling and Exception Objects

```
a = input('Number 1: ')
b = input('Number 2: ')
result = 'undefined'
try:
    result = float(a) / float(b)
except:
    typ, value, tb = sys.exc_info()
    print(f'A {typ.__name__} occurred on line {tb.tb_lineno}')
print(f'Division result is: {result}')
```

Gives info on the latest exception

A ValueError occurred on line 7

```
a = input('Number 1: ')
b = input('Number 2: ')
result = 'undefined'
try:
    result = float(a) / float(b)
except Exception as err:
    print(f'A {type(err).__name__} occurred: {err}',
                                        file=sys.stderr)

print(f'Division result is: {result}')
```

Use the Exception object to extract additional information about the error condition

A ZeroDivisionError occurred: float division by zero

student_files/ch02_files_flow_control/01_exceptions.py

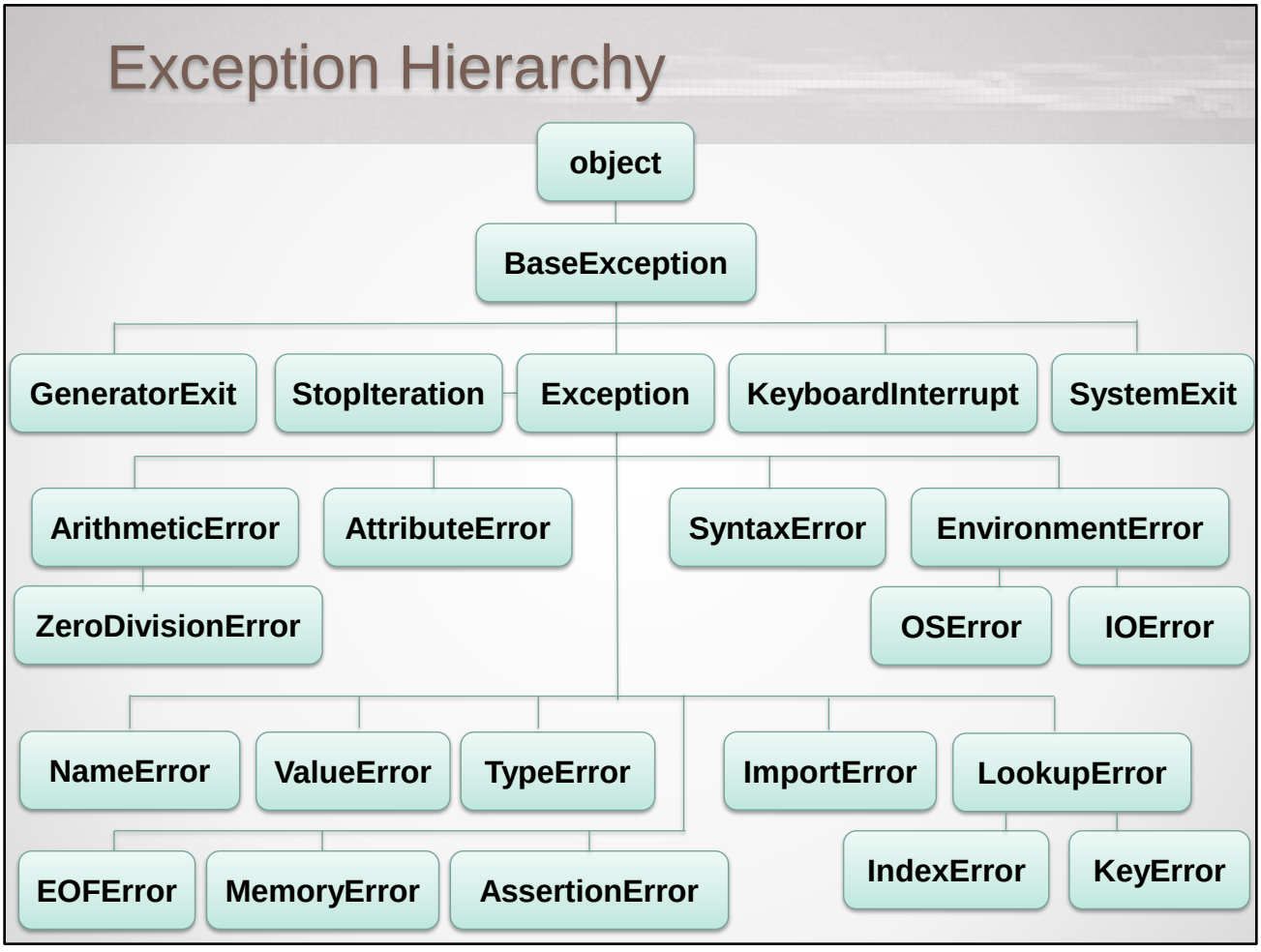
In the first example, a broad exception block is used which will capture all exception types. This can leave you wondering what happened, particularly if you must log the exceptions. The sys module provides an exc_info() method that can provide insight on the exception. It returns a tuple of values that include the exception type, the error message, and the Traceback object.

The second example is similar, but uses the exception object (err, in this case, though it can be named anything). As you can see, similar information can be extracted from the exception object as in the sys.exc_info() method. Also, the exception object contains an **args** attribute that can contain multiple messages. err can be printed directly but is really the same as err.args[0].

Note: there are situations where err.args[0] may not exist. As an example:

Fails

```
try:
    raise Exception()
except Exception as err:
    print(err)
    print(err.args[0])
```



SystemExit, GeneratorExit, and KeyboardInterrupt are NOT Exception types. These are not normally handled via a try-except control and serve other purposes, such as indicating the end of a generator or signaling the desire to shutdown the interpreter.

Numerous additional built-in exceptions exist. Refer to this document for more: https://docs.python.org/3/library/exceptions.html

Working with Files

```
open(file, mode, buffering, encoding, errors, ...)
```

- Use the built-in open() function to work with files

Default mode is *rt* (read text)

Always specify the encoding because the default encoding is platform-specific

```
# read text
file = open(filename, encoding='utf-8')

# write text
file = open(filename, 'w', encoding='utf-8')
```

mode values include = 'r', 'w', 'r+', 'a', 'x', and 'b'
(read, write, read+write, or append, create+write, binary)

f is a file object

```
f = open('myfile.txt', encoding='utf-8')
entire_file = f.read()
```

String containing entire file contents

The other arguments that may be passed into the open() method are mode='r', buffering=1, errors=None, newline=None, closedfd=True, opener=None

mode defines how a file is opened, 'r' is read, 'a' is append, 'w' is write, 'r+' is read or write, 'b' is binary, 't' is text (default). 'x' creates a file but raises a FileExistsError if the file already exists. When 'w' is used, the file is truncated to 0 bytes immediately before use.

buffering is either 1 or 0 in text mode. It is a chunk size in binary mode (e.g. 4096).

errors='strict' will raise a ValueError exception if encoding errors occur.
errors='ignore' will ignore encoding errors (and likely lead to problems). Text mode only.

encoding='utf-8', the default is platform-specific. Only valid for text mode.

newline=None or '' or '\n' or '\r' or '\r\n'. If None, universal newline support is enabled which reads platform-specific newlines in as \n and writes them out as platform-specific. If '', universal newline support is disabled. Text mode only.
closefd when set to false leaves file descriptors open after files are closed. Rarely used.
opener allows another function to serve as the open function(). Rarely used.

Reading from Text Files

- Some file object methods

```
f.readline()
my_list = f.readlines()
f.writelines(list)
```

Reads one line from a file, retains newline

Reads all lines, puts them in a list

Writes a list, one item per line to a file

- A common way of processing file data iteratively

This approach is suitable for processing large files

line will include the termination character (\n)

```
for line in open('myfile.txt', encoding='utf-8'):
    # process line from file
```

The code at the bottom of the slide works because open() returns a file object and the file object is iterable.

Note: Python recognizes 'utf-8' and 'utf8' equally! Generally, 'utf-8' is preferred.

Formal IOError Handling

If file is not found, *open()* fails, f will be *None*, and close() shouldn't be called. So, a check within the finally block is performed before closing

```
f = None
try:
    f = open('my_file.txt', encoding='utf-8')
    for data in f:
        # work with data
except IOError as err:
    print(err, file=sys.stderr)
finally:
    if f:
        f.close()
```

Optionally redirects exception messages to stderr using file=sys.stderr

This code ensures that if an error occurs while processing a file, that it is both closed and handled in an except block. If the file is not opened at all due to an error reading it, the finally will still be called but it won't actually perform the f.close() statement.

Initialization and Cleanup

- In programming, it is usually desirable to properly clean up resources when working with files
  - This is also true for other resources, such as database and socket connections, locks, and more
- The following is a common programming paradigm:

Do some initialization

Do some work

Do some cleanup

- The Python *with* control can do this

Initialization and clean up are performed internally

```
with contextmanager as obj:
    do_work
```

A *context manager* is a special object that defines how to initialize at the beginning and clean up afterwards

The ***with*** control is a somewhat complex, yet versatile structure. It requires the use of a special object called a context manager. The context manager requires two methods to be present (discussed momentarily).

Example of Using *with*

- The with control is ideally suited for working with files
  - It defines an __enter__() and __exit__() method

```
lines = []
f = None
try:
    f = open('alice.txt', encoding='utf-8-sig')
    lines = f.readlines()
except IOError as err:
    print(f'Handled {err}', file=sys.stderr)
finally:
    if f:
        f.close()
print(f'{len(lines)} lines read.')
```

These two methods are what identifies a context manager

Original version (before *with*)

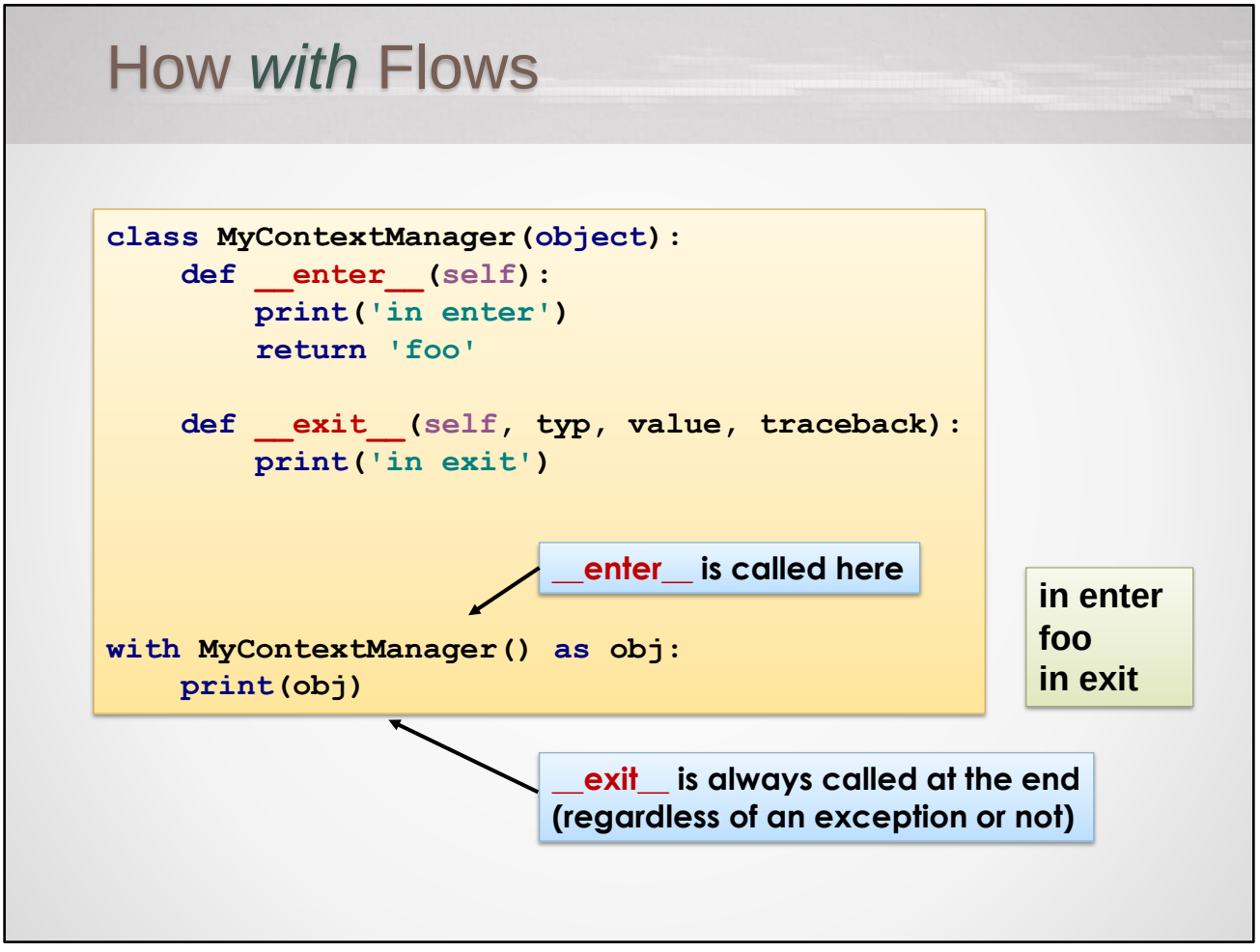
```
lines = []
try:
    with open('alice.txt', encoding='utf-8-sig') as f:
        lines = f.readlines()
except IOError as err:
    print(f'Handled {err}', file=sys.stderr)
print(f'{len(lines)} lines read.')
```

Refactored to use *with*

student_files/ch02_files_flow_control/02_ctxmgr01.py

In our examples above, we wish to read from a file. Both versions are equivalent and will result in the same output and proper file closing whether there is an exception or not. The difference is the second version uses the *with* control simplifying the code and making it easier to follow.

As a sidenote, the encoding used here (utf-8-sig) is a "signature" of utf-8 that considers the byte order mark character (\ufeff) present at the beginning of the file. This can best be viewed in the PyCharm debugger.



student_files/ch02_files_flow_control/03_ctxmgr02.py

A context manager defines both an __enter__ and an __exit__ method in a class. Though classes haven't been discussed thus far, the class concept is irrelevant to this discussion. Completely ignore the references to self at this time. When the ***with*** statement is encountered, the context manager's __enter__ method is invoked. The return value from __enter__ is passed to the **'as'** portion of the ***with*** control.

The **'as'** portion of the control is optional but receives the return value from __enter__().

When the block within the ***with*** control is finished executing, the __exit__() method will be invoked. The __exit__() will be called no matter whether an exception is raised or not.

Writing to Text Files

- In write 'w' mode, use the file's write() method
  - write() accepts a string that should end with \n
  - Python will translate \n into a platform-specific line termination character

```
data = [
    'Lorem ipsum dolor sit amet, consectetur ',
    ...
    'qui officia deserunt mollit anim id est laborum.',
]

try:
    with open('data.txt', 'w', encoding='utf-8') as f:
        for item in data:
            print(item, file=f)
except IOError as err:
    print(err, file=sys.stderr)
```

student_files/ch02_files_flow_control/04_writing.py

This example takes a list of strings and writes it to a file.

Simple *Diff* Utility with Multiple Context Managers

More than one context manager can be defined in a single with control

sample1.txt

this is my
sample
file
but
it is different
than
the the
other
file.

sample2.txt

this is my
sample
file
but
it is different
than
the
other
file.

```
file1, file2 = 'sample1.txt', 'sample2.txt'

with open(file1, encoding='utf-8') as f1,
     open(file2, encoding='utf-8') as f2:
    for line_num, (line1, line2) in enumerate(zip(f1, f2), 1):
        line1 = line1.strip()
        line2 = line2.strip()
        if line1 != line2:
            print(f'Diff line: {line_num:<10}|{line1:40}|{line2:40}|')
```

Diff line: 7 |the the |the |

```
# Using the Python standard library module...
from difflib import Differ
diff = Differ().compare(open('sample1.txt', encoding='utf-8').readlines(),
                        open('sample2.txt', encoding='utf-8').readlines())
print(*diff)
```

student_files/ch02_files_flow_control/05_diff_util.py, sample1.txt, sample2.txt

The slide shows two examples. The first is our home-grown, simple *diff* utility. It uses a *with* control to work with two context managers simultaneously. Within the *with*, the zip() function iterates over two files at once. The lines are compared, and an output is generated only when the lines differ.

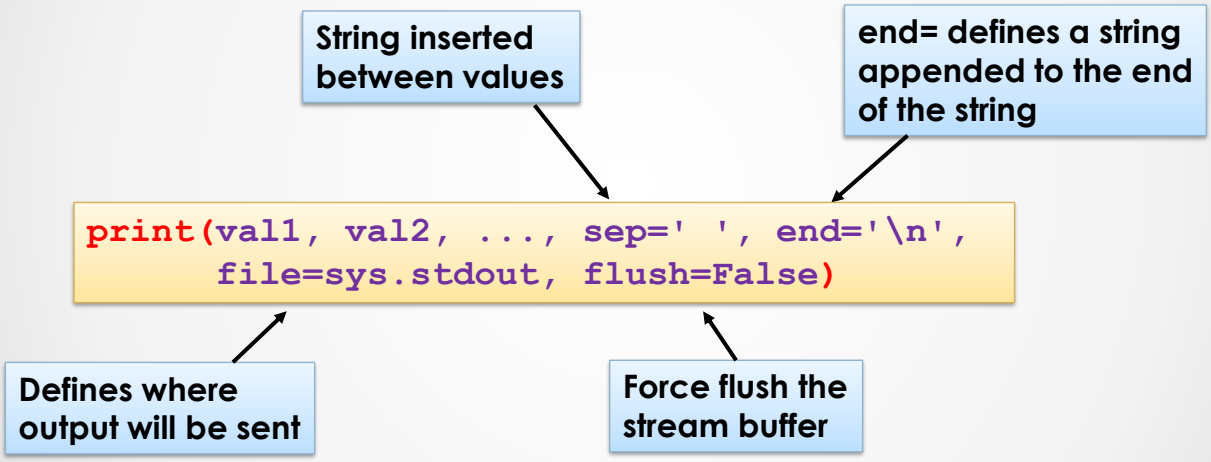
The bottom of the slide illustrates the Python standard library module, **difflib**, and its helper function, **compare()**.

It compares the lines from the file and indicates lines that differ. A minus (-) indicates the different line in file 1 (left file), while a plus (+) indicates the different line in file 2. The results of the compare() method are shown to the right.

this is my
sample
file
but
it is different
than
- the the
+ the
other
file.

The *print()* Function

- The print function has the following syntax



Your Turn! - Task 2-1

- Write a program to *find the top 10 baseball salaries* for a specified input year (1985 - 2016)
- File output format should be as follows

```
Search salaries for what year?--> 1985
Results
Name                                Salary              Year
Mike Schmidt                        $2,130,300.00       1985
Gary Carter                         $2,028,571.00       1985
George Foster                       $1,942,857.00       1985
Dave Winfield                       $1,795,704.00       1985
Rich Gossage                        $1,713,333.00       1985
Dale Murphy                         $1,625,000.00       1985
Jack Clark                          $1,555,000.00       1985
Bob Horner                          $1,500,000.00       1985
Eddie Murray                        $1,472,819.00       1985
Rickey Henderson                    $1,470,000.00       1985
```

Examine Salaries.csv and People.csv to understand the file formats

First and last name are found in ***People.csv***

Salary and year are in ***Salaries.csv***

Follow additional instructions within ch02_files_flow_control/task2_1_starter.py

Data files can be found in the student_files/resources/baseball folder.

Helpful hints:
To find the largest salaries, sort it by salary. To do this you can use .sort(key=???) where ??? is a function that returns the field representing the salary. Since functions are an upcoming topic, the sort function has been provided for you.

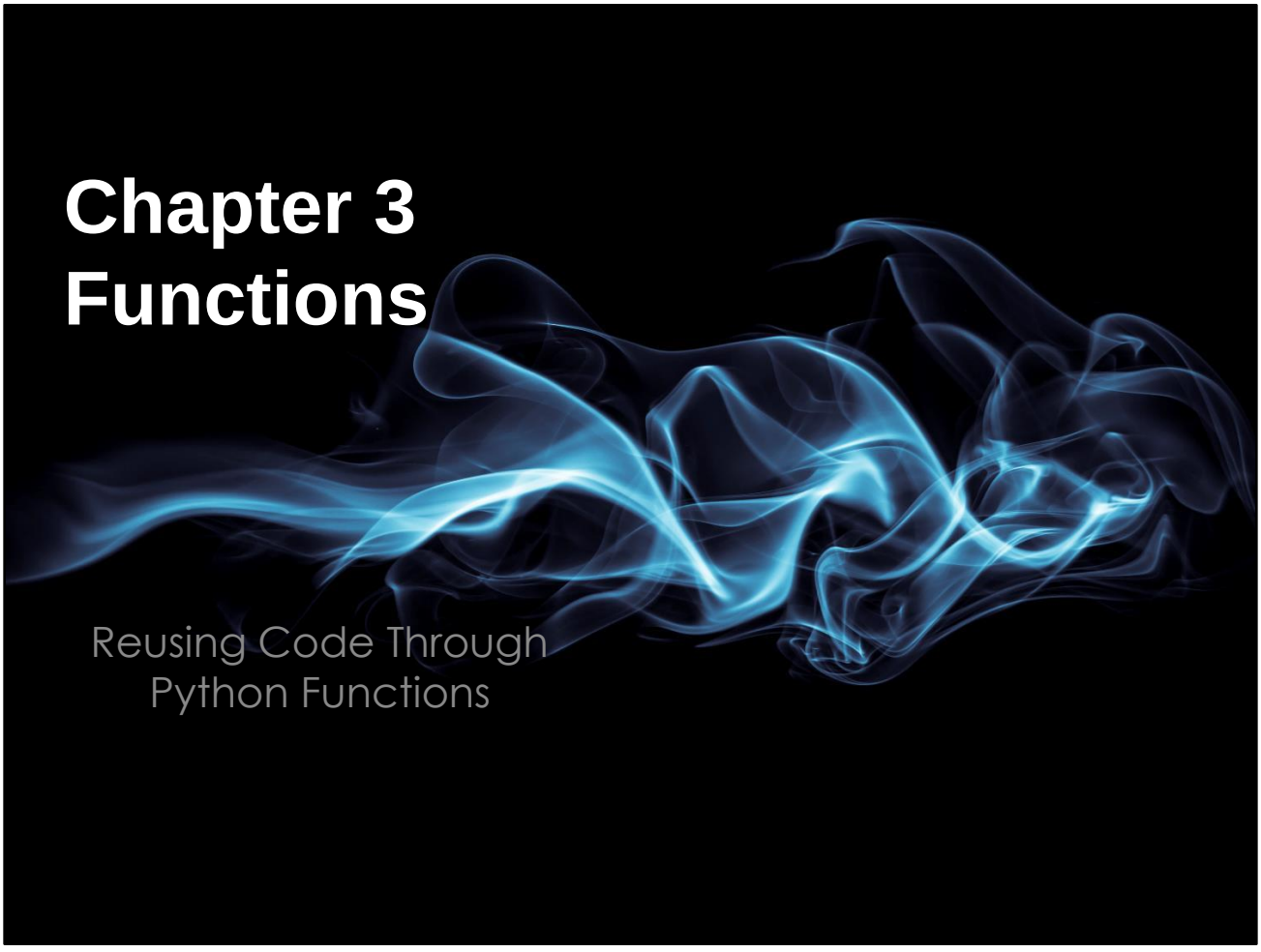
```
salaries.sort(key=sort_func)
```

Source for baseball statistics: http://seanlahman.com/baseball-archive/statistics/.

Chapter 2 - Summary

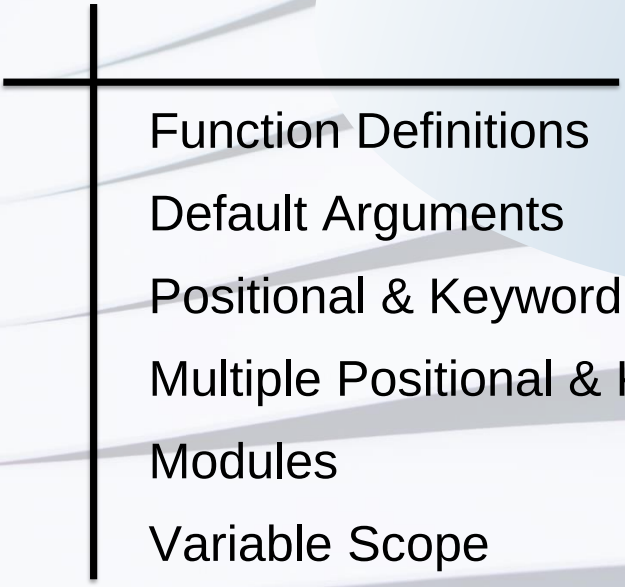
- Python provides *exception handling* capabilities but leaves it up to the developer to implement it
- Prefer the *with* control when working with files as this will ensure files are automatically closed
- In Python 3, specify the *encoding* when opening files since the default value is platform specific
- The *print()* function in Python 3 provides extra options such as the ability to control how values are separated, end of line characters, and where output is routed

Chapter 3
Functions



Reusing Code Through
Python Functions

Chapter 3 - Overview



- Function Definitions
- Default Arguments
- Positional & Keyword Args
- Multiple Positional & Keyword Args
- Modules
- Variable Scope

Defining and Calling Functions

- Functions must be defined *before* they can be called

This function declares that two arguments need to be passed in

```
def summary(customer, amount):
    return f'Customer: {customer.get("first")} \
            {customer.get("last")}, amount: ${amount:,.2f}'


cust = {
    'first': 'James',
    'last': 'Smith'
}

results = summary(cust, 1108.23)
print(results)
```

The contents of a function will be indented

Return values are optional. A value of *None* is returned when a return statement is omitted

Customer: James Smith, amount: $1,108.23

student_files/ch03_functions/01_functions.py

Functions must be defined (or imported) before they can be called. All parameters declared by the function need to be filled in when calling the function.

Using Default Arguments

```
def word_counter(filepath, min_wordsize=1, max_results=10,
                 encoding='utf-8-sig'):
    word_dict = defaultdict(int)
    with open(filepath, encoding=encoding) as f:
        for line in f:
            words = line.strip().split()
            for word in words:
                if len(word) >= min_wordsize:
                    word_dict[word] += 1

    sorted_dict_items = sorted(word_dict.items(),
                               key=lambda kv: kv[1], reverse=True)
    return sorted_dict_items[:max_results]
```

A *defaultdict* cannot produce a KeyError

```
if __name__ == '__main__':
    sample_file = '../resources/gettysburg.txt'
    results = word_counter(sample_file)
    print(results)
```

Other ways to call the function:

```
word_counter(sample_file, 3)
word_counter(sample_file, 5, 20)
```

student_files/ch03_functions/count_module.py

An explanation of this code:
This function is a modularized version of our Task 1-4.

The purpose of this example is to see the effects of default arguments. Notice that the min_wordsize parameter has a default argument set to 1, therefore by default all words are returned from the desired file, but this can be changed by passing a value into the second argument. The max_results parameter has a 10 value by default, but this value can be changed too.

At the bottom, the word_counter() function is invoked different ways illustrating the use or default values.

For the specific details of this function, see the code in the student files.

Keyword Arguments

- All functions in Python support keyword arguments
  - Benefits of keywords include:
    - Better code readability
    - Decreased dependency on parameter order

```
def word_counter(filepath, min_wordsize=1, max_results=10,
                 encoding='utf-8-sig'):
    ...
```

All of these are valid calls

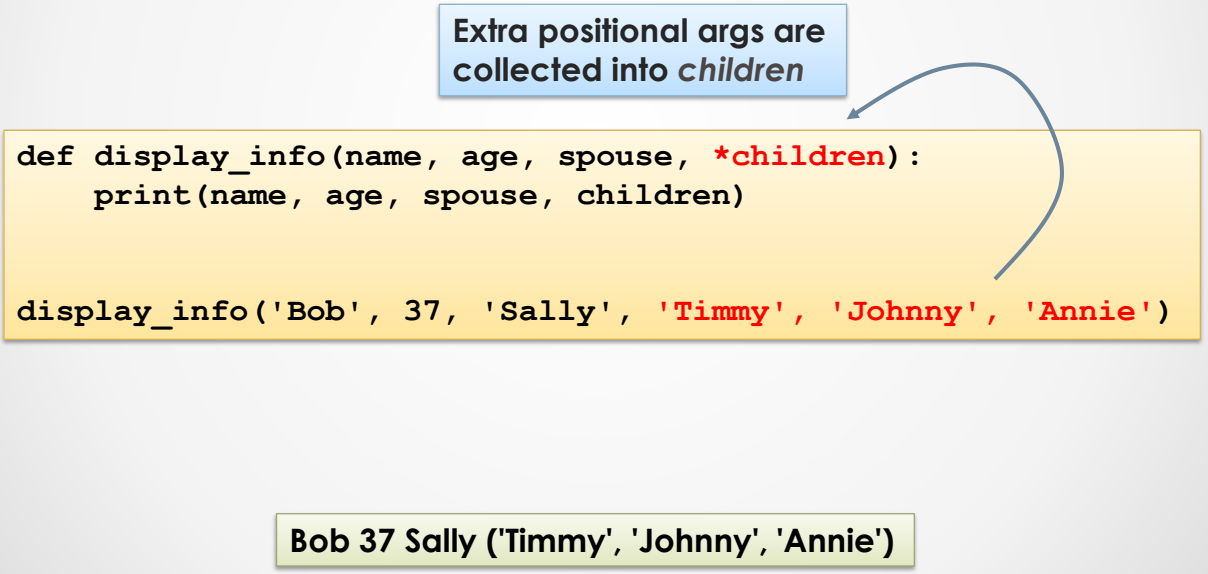
```
word_counter(sample_file)
word_counter(sample_file, 4, 15)
word_counter(filepath=sample_file, min_wordsize=4, max_results=15)
word_counter(sample_file, encoding='utf-8')
word_counter(max_results=15, filepath= sample_file, min_wordsize=5)
word_counter(max_results=15, filepath=sample_file)
```

student_files/ch03_functions/count_module.py

The first two function calls rely solely on positional and default arguments. The third call uses keyword arguments and defaults. The remaining calls demonstrate various ways that keyword arguments can be used to call this function.

Multiple Positional Arguments

- If the number of parameters to a function may be unknown, use the (*) character



student_files/ch03_functions/02_multiple_args.py

Sometimes it is unknown how many arguments are required for processing within a function. Python provides a way to pass as many arguments as needed using the (*). For this to work properly, the *children argument MUST be supplied last. It cannot appear before any other positional arguments and only ONE multiple positional argument can be supplied.

Multiple Keyword Arguments

- It is also possible to supply multiple keyword parameters
  - Results are placed into a *dictionary*

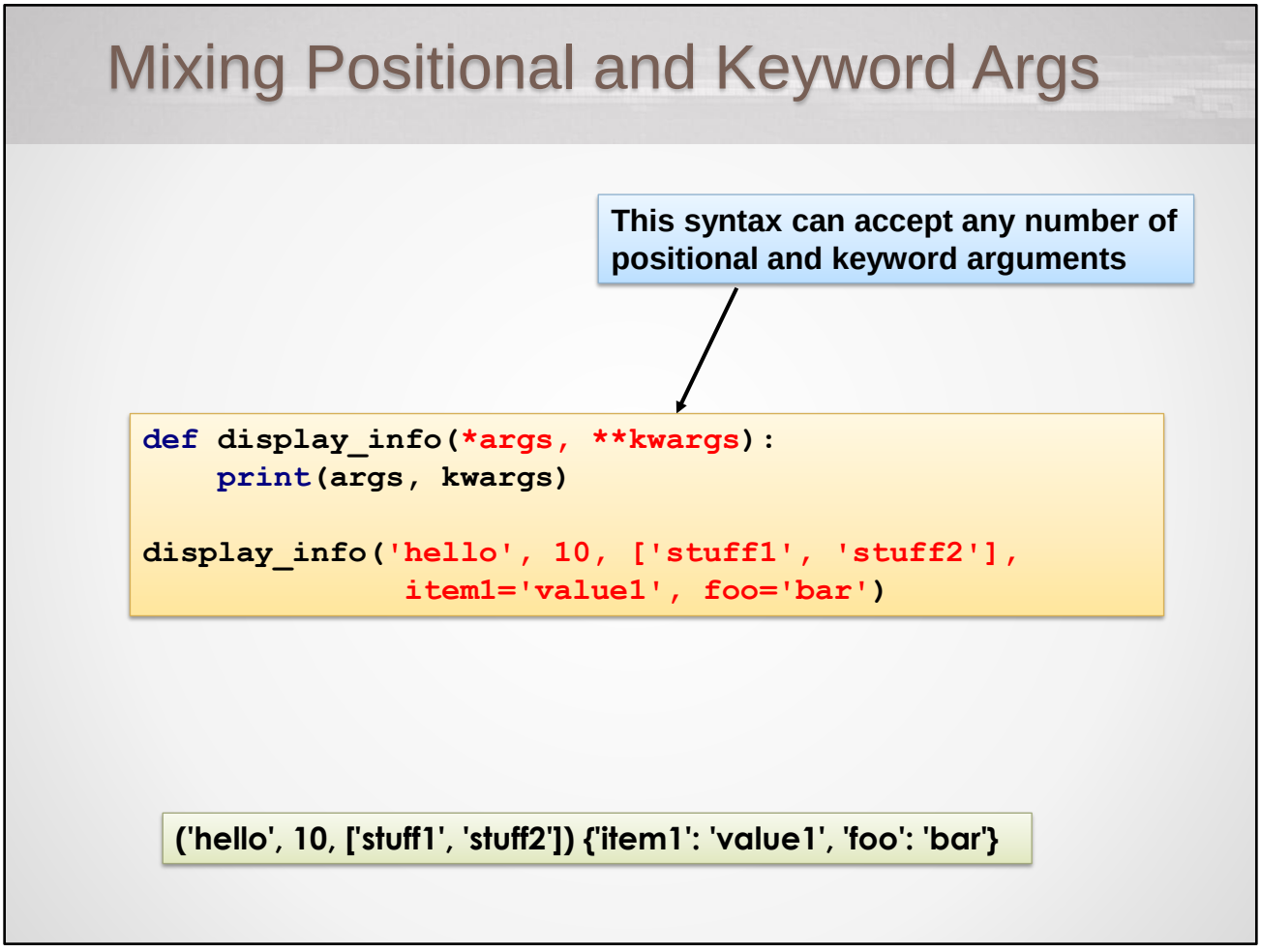
```
def display_info(**family):
    print(family)


display_info(name='Bob', age=37, spouse='Sally',
             child1='Timmy', child2='Johnny', child3='Annie')
```

Keyword args are collected into *family*

```
{'child1': 'Timmy', 'child2': 'Johnny', 'child3':
'Annie', 'name': 'Bob', 'age': 37, 'spouse': 'Sally'}
```

student_files/ch03_functions/02_multiple_args.py

Mixing Positional and Keyword Args



The use of *args, **kwargs is common in Python, particularly under specific scenarios (such as decorators, functions that take large numbers of arguments, etc.).

Unpacking Arguments

- In a function definition, * and ** 'collect' values
- These symbols may also be used in the *function call*
  - Values are dispersed, or *spread out* in the function def
  - It works the opposite as it does in the function declaration

```
def display_results(customer, purchase_amount):
    print('Customer: {first} {last}, amount: ${p_amt:,.2f}'
          .format(p_amt=purchase_amount, **customer))

cust = {
    'first': 'James',
    'last': 'Smith'
}

display_results(cust, 1108.23)
```

*customer is a dictionary. Using ** on the dictionary causes the dictionary to be **expanded** into keyword arguments in the call to format()*

student_files/ch03_functions/02_multiple_args.py

Introducing Modules

- *word_counter()* exists in its own .py file and can be imported into other .py files
  - These files are called modules
- Modules, .py files, serve as both holders of code and namespaces in Python
  - Declare functions, variables, & classes within modules
  - These *top-level* items are called *attributes* and must be imported before they can be used
    - *word_counter()* is an *attribute* of count_module

```
import sys

import count_module

print(sys.version)
print(count_module.word_counter('gettysburg.txt))
```

Module names now serve as *namespaces*

Types of Modules

- There are three categories of modules used in Python

Standard Library modules

These ship with Python, are always available for import, and can be found in the <PYTHONHOME>/lib directory
Examples: os, math, re

Third-party modules

These are added into Python later, typically using pip (see next slide), and are usually found in <PYTHONHOME>/lib/site-packages
Examples: requests, numpy, pandas

Custom (your) modules

These are created by you, placed into a directory of your choice that's added to the PYTHONPATH environment variable

Installing Third-Party Tools Using *pip*

- Third-party tools are usually installed using pip (python installation package) tool

To use pip:
Ensure <PYTHONHOME>/Scripts directory is on your path (Windows)
<PYTHONHOME>/bin (Linux/Unix)

  - Pip syntax: pip *command* [options]
  - Commands include:

install	config
download	search
uninstall	cache
freeze	index
inspect	wheel
list	hash
show	completion
check	debug
	help

Most Python distributions/ versions already ship with the *pip* tool

Other tools, not discussed here, exist for installing packages into Python including easy_install (older), wheel, pipenv (for virtual environments), conda (for Anaconda distributions), poetry (similar to pipenv).

What Is PyPI?

- Examples using pip:

pip install *package*	pip install *prettytable*
python -m pip install *package*	Alternate way to run pip
pip3.13 install *package*	pip3.13 install *prettytable*
pip uninstall *package*	pip3.13 install *prettytable*

- PyPI is the *Python Package Index Repository*
- Contains third-party resources (packages)
  - Sometimes called the *Cheese Shop*

(named after a Monty Python skit)

`http://pypi.python.org/pypi`

The Python Package Index, pronounced Py-pee-eye, contains almost 500000 third-party tools (projects) for use or download with varying licensing, documentation, support, etc.

Other Ways to Import Items

- There are several techniques for importing items

```
import numpy as np
```

Imports everything from numpy, but places it into the namespace aliased as np

```
from os import listdir
```

While the entire module is still read, a variable is created only for the single item (listdir)

```
from os import listdir as get_files
```

The entire os.py file is read, listdir is imported but a variable called get_files is created and points to listdir

To determine the contents of a module, issue the dir(*modulename*) command after importing the module in a Python shell.

Adding Your Own Modules

- Python programs locate modules by looking at the sys.path variable
- To ensure sys.path "sees" your modules, perform one of these:
  1. Modify sys.path directly
     sys.path.append('/home/modules')
  2. Put your modules in
     <PYTHON_HOME>/Lib/site-packages
  3. Create an OS-level environment variable called PYTHONPATH **Preferred Way**

The third option is generally the preferred option. While it is possible to manipulate the sys.path variable directly, this has to be done within the program source code and therefore has to be maintained internally to the code. Despite the name *site-packages*, this location typically contains third-party (installed) modules. Most Python distributions will automatically have this directory defined on their sys.path, so it could be used for internal modules, but typically this is not the case.

The best, most flexible option is to simply set an environment variable. This can be done as follows:

In Windows:
(cmd shell) set variable=value (set PYTHONPATH=c:\modules) <-- 1 session only
or...
Control Panel > System > Advanced System Settings > Advanced Tab > Environment Variables > User Variable > New... > PYTHONPATH and c:\modules

Mac:
Open a Text Editor, File -> Open, locate your .profile or .bashrc or .bash_profile file in your home directory. Add an entry to it as follows:
export PYTHONPATH=~/modules
(Note: if the file doesn't exist, you may create it.)

Using Custom Modules *(in Three Steps)*

To invoke the *word_counter()* function from another module (such as temp.py)

1 **Set the PYTHONPATH to contain our custom module directory**

On Windows (from a command window):

```
set PYTHONPATH=.;%PYTHONPATH%;<path_to_student_files>\ch03_functions
```

In Linux and Mac OS X (from a terminal window):

```
export PYTHONPATH=.:$PYTHONPATH:<path_to_student_files>/ch03_functions
```

2 **Add the following to student_files/temp.py:**

```
from count_module import word_counter
print(word_counter('resources/gettysburg.txt'))
```

3 **Run `python temp.py` from the student_files directory in the command or terminal window**

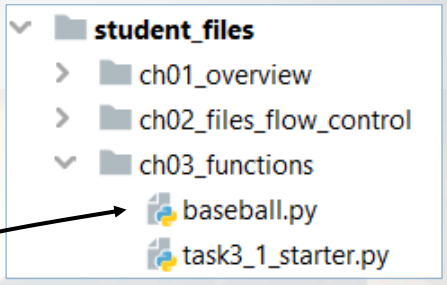
To import a module, Python must be able to find it. It searches for modules according the values found in sys.path. sys.path will contain some predefined directories (like the PYTHONHOME/lib directory) as well as directories you have defined on the PYTHONPATH environment variable.

Step 1 above adds the location of our custom module onto the PYTHONPATH. Note: we also could have added our student_files directory onto the PYTHONPATH, but this affects the import statement (e.g., now it would be from ch03_functions.count_module import word_counter).

Step 2 imports the new class now that it can be found by the interpreter.

Step 3. Runs the script.

Your Turn! - Task 3-1



- Refactor the baseball exercise (Task 2-1) to use functions and modules
  - Work from the module called baseball.py
  - Move code related to loading the data into a function called *load_data()* within baseball.py
  - Import baseball.py from *task3_1_starter.py* and invoke load_data() to retrieve data into the starter file

baseball.py

load_data ()

→

task3_1_starter.py

import baseball (should work), or
import ch03_functions.baseball as baseball

Follow additional hints within ch03_functions/task3_1_starter.py

Note on imports: If the current directory is on your sys.path variable (denoted with just an empty string ('') or you are running the code from within PyCharm, then `import baseball` will work.

As a side note, the student_files directory should be placed on our PYTHONPATH when running the code in another IDE or from a terminal (command prompt). If you are using PyCharm, it will be placed on your sys.path automatically so you do not need to create/modify a PYTHONPATH environment variable.

Variable Scope

- Variable scope rules follow the acronym LEGB

```
x = 10
def f1():
    x = 20
    def f2():
        x = 30
        print(x)
    f2()
f1()
```

Global x

Enclosing x

Local x

This is the local value of x and therefore will be printed

You might expect this to change the global *num_records*, but it doesn't, it creates a local variable

[0, 1, 2, 3, 4]

```
values = list(range(100))
num_records = 5

def change_records():
    num_records = 10

def display_records():
    print(values[:num_records])

change_records()
display_records()
```

student_files/ch03_functions/03_legb.py and 04_assigning.py

LEGB is an acronym often used to help describe the scoping rules of Python. It stands for Local, Enclosing, Global, Built-in.
In the top example, removing the innermost value of x will then print the x=20 value, removing this value of x will print the x=10 value.

Enclosing functions, or the placement of other functions inside of other functions, are discussed in the Intensive Intermediate Python course.

In the second example above, the reason num_records doesn't change the globally defined variable is because a local variable is created instead due to the assignment operator.

Accessing Global Vars

- To *modify* a global variable within a function, use global

Changes the global variable num_records

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
values = list(range(100))
num_records = 5

def change_records():
    global num_records
    num_records = 10

def display_records():
    print(values[:num_records])

change_records()
display_records()
```

- A built-in function called globals() also provides access to global variables

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
def change_records():
    globals()['num_records'] = 10

def display_records():
    print(values[:num_records])

change_records()
display_records()
```

student_files/ch03_functions/04_assigning.py

global creates a reference to a globally declared variable, allowing for manipulation (modification) of the global from within the function. globals() is a dictionary that returns all global vars.

Use of global and globals() should be limited and only taken advantage of when necessary.

Not mentioned or used here is another function, called locals() which provides dictionary-style access to all local variables of a function.

Passing Arguments

- Immutable values passed to a function are passed by value while mutable values are passed by reference

```
def update_values(v1, v2):
    v1 *= 2
    v2 *= 2


values1 = (1, 2)
values2 = [3, 4]
update_values(values1, values2)

print(values1, values2)
```

(1, 2) [3, 4, 3, 4]

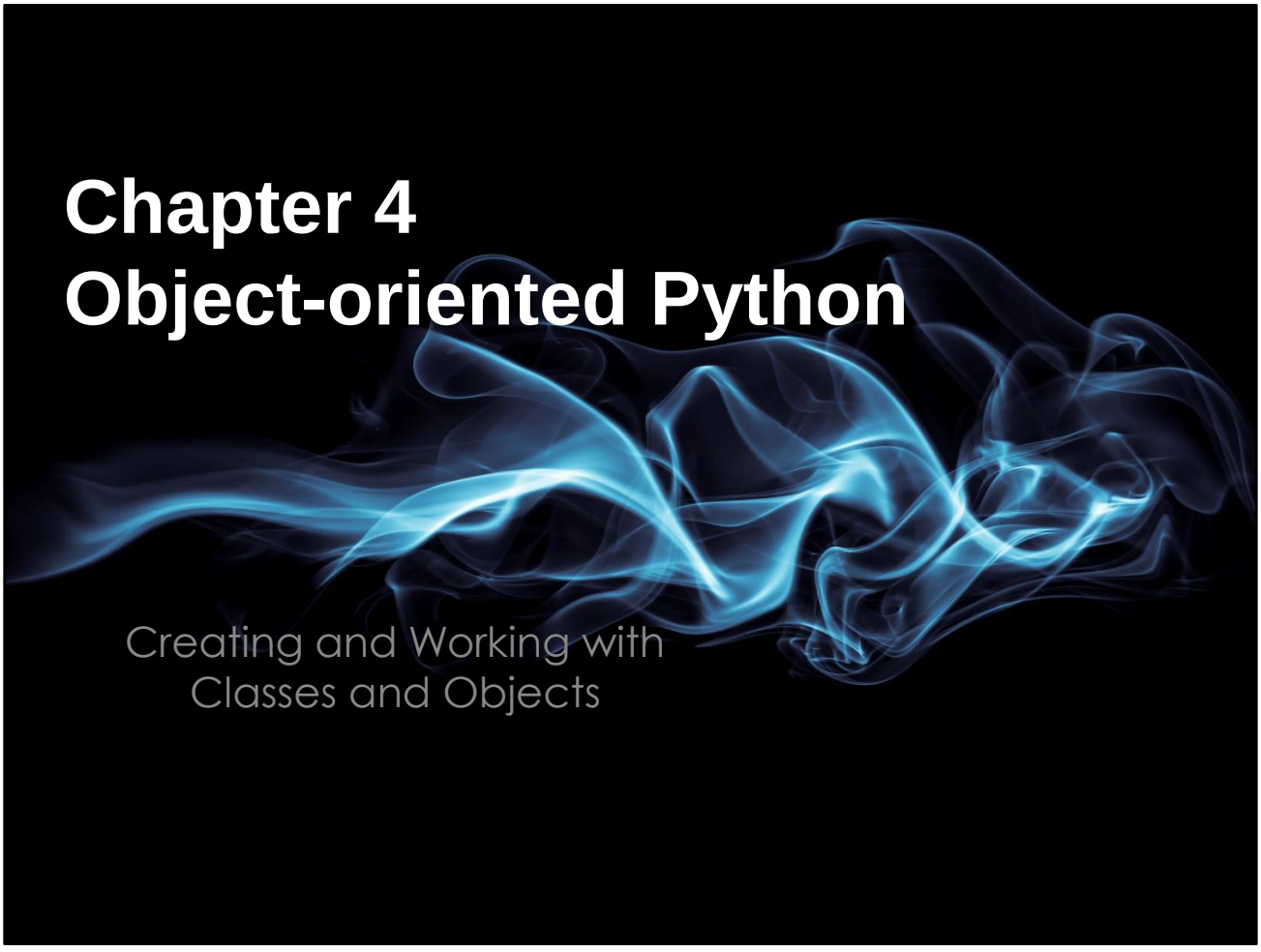
This list gets modified, the tuple does not

student_files/ch03_functions/05_passing_args.py

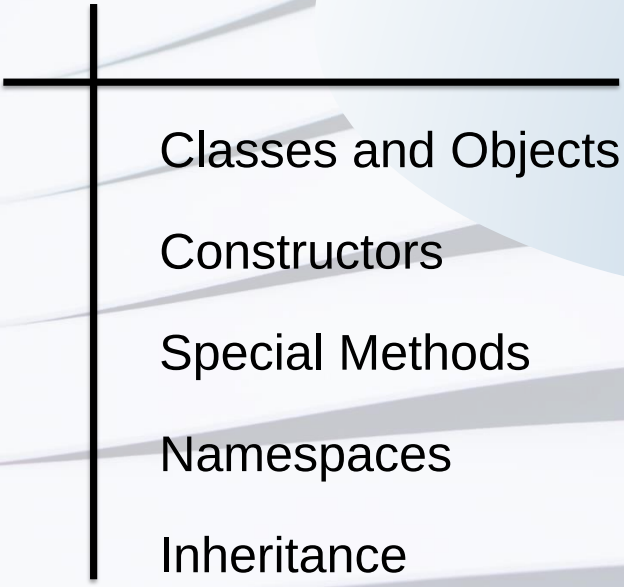
In this example, two values are passed to the *update_values()* function. One is immutable (the tuple). The other is mutable (the list). Inside the function each value is modified. At the end, the variables passed in are displayed. Notice the list has been modified, but the tuple is unchanged.

Chapter 3 - Summary

- Functions can support default arguments, keyword arguments, multiple positional arguments, multiple keyword arguments and unpacking of variables into individual vars
- Ensure custom modules are properly placed onto the PYTHONPATH
- Variables are either local or global. Use of globals should be minimized



Chapter 4 - Overview



- Classes and Objects
- Constructors
- Special Methods
- Namespaces
- Inheritance
- Properties

Why Use Classes in Python?

- Earlier we introduced named tuples

```
from collections import namedtuple

Contact = namedtuple('Contact', 'first last age email')

c = Contact('John', 'Smith', 43, 'jsbrony@yahoo.com')
```

- Limitations include:
  - Inability to easily modify data: it is effectively a tuple
  - Operations on this data are maintained separately

- Classes create (instantiate) objects
  - Object data not only can be modified, but can use the behaviors (methods) defined in the class

Classes provide flexibility that other data structures do not. For example, the ability to associate methods (operations) with related data is an advantage. Inheritance provides another benefit.

Class Basics

```
class Contact:
    """ Defines a Contact type """
    def __init__(self, name='', address=''):
        self.name = name
        self.address = address

    def __str__(self):
        return self.name


c = Contact('John Smith', '123 Main St.')
print(c.name)
print(c, type(c))
```

```
John Smith
John Smith <class '__main__.Contact'>
```

student_files/ch04_oo/01_basics.py

This example illustrates a Python class. It contains a couple of magic methods, __init__ and __str__. These will be discussed shortly. The class is used to "instantiate" an object. The instantiation occurs on the line: c = Contact(). This creates an object which, after a type check, shows that it is a Contact type.

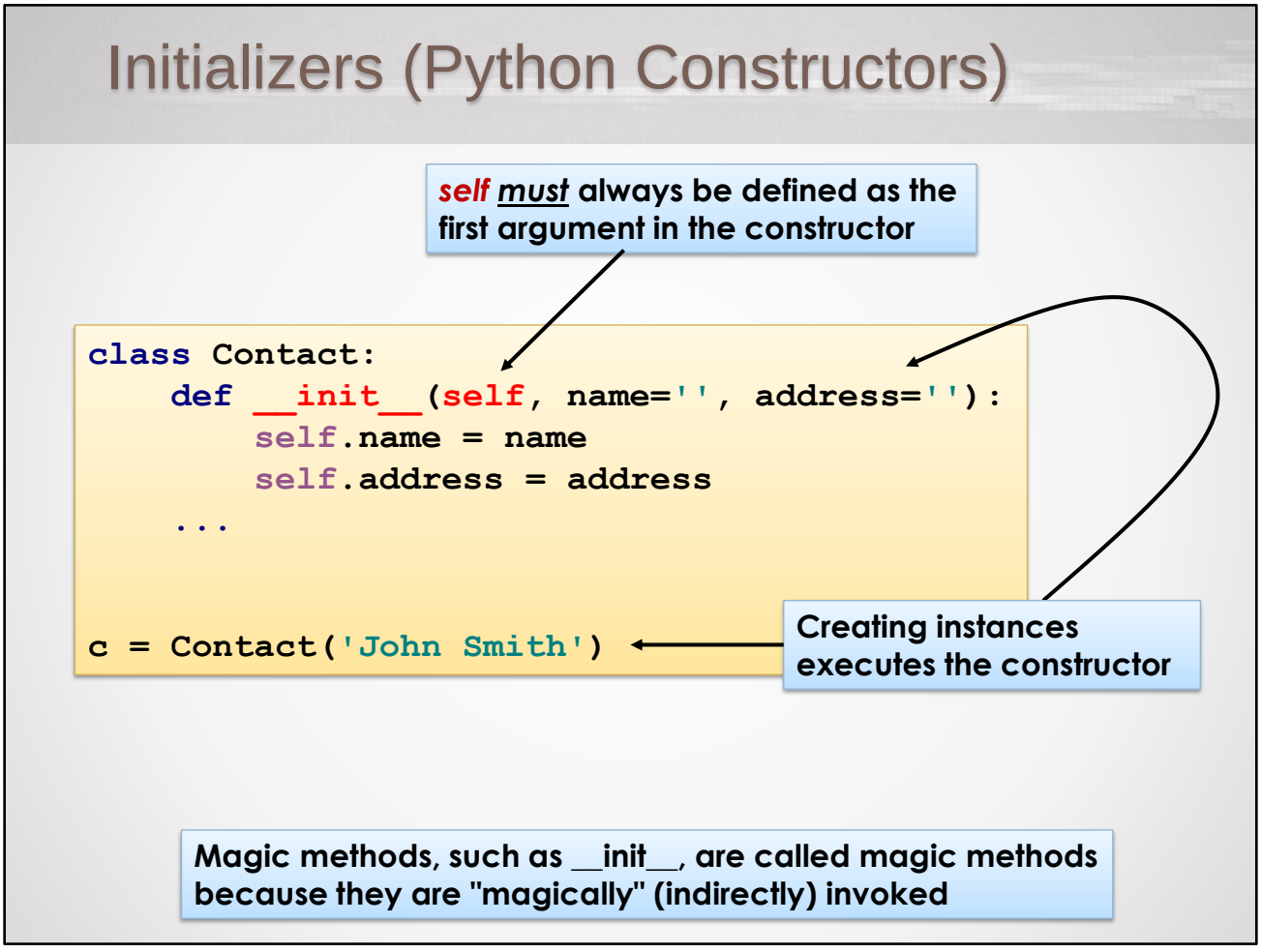
Note that in other languages you might be tempted to use the new operator. But Python doesn't define a new operator. So this statement:
c = new Contact( )

is actually

c = Contact( ) in Python

Notice that properties such as alt_email can be added to it at any time.

Initializers (Python Constructors)



student_files/ch04_oo/01_basics.py

Though the __init__() magic method is short for initializer, it is also sometimes referred to as a constructor due to its similarities to constructors in other object-oriented languages.

Class-Based Word Counter

```
class WordCounter:
    def __init__(self, filepath, min_wordsize=1, max_results=10,
                 encoding='utf-8'):
        self.word_dict = defaultdict(int)
        self.filepath = filepath
        self.min_wordsize = min_wordsize
        self.max_results = max_results
        self.encoding = encoding

    def results(self):
        with open(self.filepath, encoding=self.encoding) as f:
            for line in f:
                words = line.strip().split()
                for word in words:
                    if len(word) >= self.min_wordsize:
                        self.word_dict[word] += 1
        sorted_dict_items = sorted(self.word_dict.items(),
                                   key=lambda kv: kv[1], reverse=True)
        return sorted_dict_items[:self.max_results]

sample_file = '../resources/gettysburg.txt'
counter = WordCounter(sample_file, min_wordsize=5)
print(counter.results(), counter.word_dict)
```

Special (magic) methods

Instance methods

Instance variables are unique to each object

Instantiation

student_files/ch04_oo/count_module2.py

For brevity, exception handling and a few other lines of code have been omitted. Refer to the source file for the complete solution. While it's arguable as to whether a class is needed for the current example, it could be expanded to add other methods, such as a *save_results()* or a *filter()* method. Without a class, grouping these operations into a cohesive unit would have to be done at the module level, which can feel less cohesive.

This class behaves like our Task 1-5. It contains a two methods, __init__() and results(). The class is used to *instantiate* an object (called counter). The instantiation occurs on the line: counter = WordCounter(...).

In other languages, you might be tempted to use the `new` operator. But Python doesn't define a *new* operator. So, the statement, **counter = new WordCounter(*filename*)** would actually be **counter = WordCounter(*filename*)** in Python.

Notice that Python doesn't define anything as public or private. Instead, everything is visible outside of the class, including word_dict. Because everything is publicly visible, it is an anti-pattern in Python to create getter and setter methods.

Public and Private Attributes

- Python does not provide a public or private capability
  - Everything in a class is public (except local variables)
  - Everything is accessible outside of the class
- In some OO languages, we make fields private and then generate getter and setter functions to access the fields
  - In Python, this is an *anti-pattern*
  - However, getters and setters in those languages do provide value by checking inputs or formatting values
  - Since this capability is useful, Python provides its version: *properties*

Defining Properties

Properties are Python's version of "getters" and "setters"

```
class Contact:
    def __init__(self, name='', address='', email=''):
        self.name = name
        self.address = address
        self.email = email

    def __str__(self):
        return f'{self.name} {self.address}'

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, email):
        self._email = ''
        if '@' in email:
            self._email = email

c = Contact('Bob', '123 Main St.', 'bobmail')
print(f'Bob\'s email: {c.email}')
c.email = 'bob@yahoo.com'
print(f'Bob\'s email: {c.email}')
```

Here the setter gets called

Use @property to define a method that can act like a getter

@prop.setter is the syntax to define a setter

Invokes email() getter

Invokes email() setter

student_files/ch04_oo/02_properties.py

Notice that within the constructor the setters can still be invoked. This way, if a Contact object is created and an invalid email is passed into it, it will still invoke the validation within the setter.

Properties provide a means to perform setter, getter, and even deleter interactions. The decorator syntax is as follows:

```
@property
def foo(self):
    return self._foo

@foo.setter
def foo(self, foo):
    self._foo = foo

@foo.deleter
def foo(self):
    del(self._foo)
```

Your Turn! - Task 4-1

- Examine the WordCounter class
- Convert the results() function into a read-only property (i.e., it has no setter)
  - Modify the class such that results() is accessed as a property
  - What advantages does a property provide here?
- Convert the min_wordsize attribute into a property
  - Ensure that a *min_wordsize can't be zero or negative*

Follow additional hints within ch03_functions/task4_1_starter.py

Class Attributes

- Variables created at the class level are called *class attributes*
  - They should be modified via the class:

```
class RaceCar:
    MAX_SPEED = 245
    ACCELERATION = 5

    def __init__(self):
        self.speed = 0

    def accelerate(self):
        self.speed += RaceCar.ACCELERATION
        if self.speed > self.MAX_SPEED:
            self.speed = self.MAX_SPEED
```

Class attributes can be accessed by both the class *and* the instance, but should not be modified by the instance

Note: do not try to *modify* ACCELERATION through the instance, as in self.ACCELERATION = 10 as this would not work in the desired way

student_files/ch04_oo/03_class_attributes.py

As the note at the bottom indicates, you should be sure not to modify the class attribute through the instance as shown. This will cause the instance to be given its own acceleration attribute while the class level attribute will still exist separately and unchanged.

Inheritance: Classic Constructor Call

```
class Contact:
    def __init__(self, name='', address='', phones=None):
        self.name = name
        self.address = address
        self.phones = phones

    def __str__(self):
        return f'{self.name} {self.address} {self.phones}'

class BusinessContact(Contact):
    def __init__(self, name='', address='', phones=None,
                 email='', company='', position=''):
        Contact.__init__(self, name, address, phones)
        self.email = email
        self.company = company
        self.position = position


bc = BusinessContact('John Smith', '123 Main St.',
                     {'home': '(970)455-2390'})
print(bc)
```

Call to object's base class constructor is not necessary

Calls the base class constructor

student_files/ch04_oo/04_inheritance_classic.py

With inheritance, include the base class in parentheses following the name of the class. The class at the top of the hierarchy does not require the object class to be specified.

To ensure that instance attributes are properly set, your derived class should call the base class constructor. There are 3 ways to do this (this and next slide illustrate this). The classic approach is shown here. This approach has the major disadvantage of having to reference the superclass name within the subclass.

Inheritance: Using super() - *Preferred*

```
class Contact:
    def __init__(self, name='', address='', phones=None):
        self.name = name
        self.address = address
        self.phones = phones

    def __str__(self):
        return f'{self.name} {self.address} {self.phones}'

class BusinessContact(Contact):
    def __init__(self, name='', address='', phones=None,
                 email='', company='', position=''):
        super().__init__(name, address, phones)
        self.email = email
        self.company = company
        self.position = position


bc = BusinessContact('John Smith', '123 Main St.',
                     {'home': '(970)455-2390'})
print(bc)
```

Uses super() class constructor

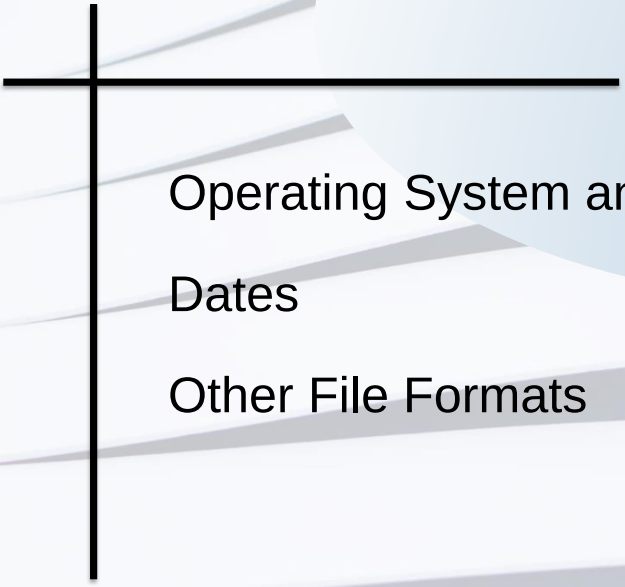
student_files/ch04_oo/05_inheritance_super.py

When using super(), be sure *not* to pass self. It is implicit within the super() call.

Chapter 4 - Summary

- Classes in Python have numerous differences from classes found in other OO languages
  - Python classes behave as a kind of namespace
  - *self* is not implicit, but must be declared as the first argument to a method
  - There are *no public and private* keywords
  - Properties can be defined to provide setter and getter capabilities

Chapter 5 - Overview



Operating System and Environment

Dates

Other File Formats

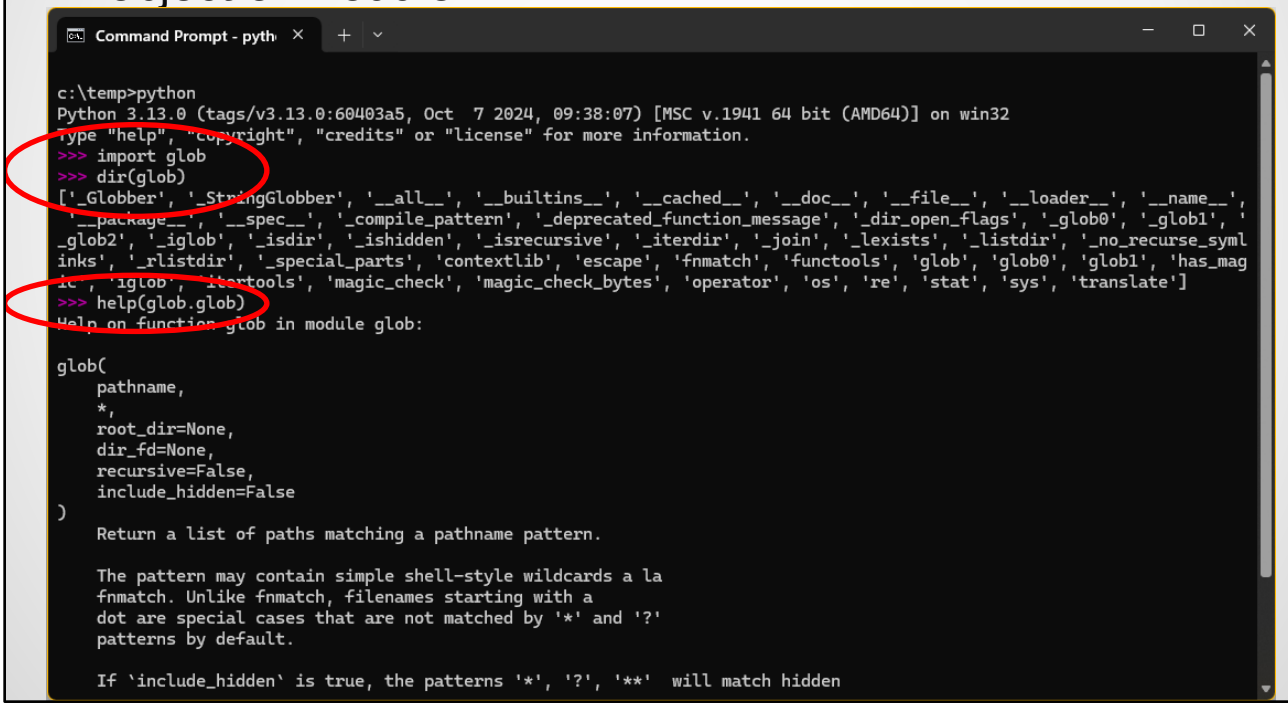
Introducing the Python Standard Lib

- The Python Standard Library contains over 200 modules containing various functions and classes
- Documentation **https://docs.python.org/3/library/index.html**
  - The standard library contains modules that assist with

Working with the operating system
Working with the Python environment
Using Time and Date Objects
Task Scheduling
Logging
Mathematical and Statistical Tools
Working with JSON
Regular Expressions
XML
Subprocesses
Threading
Multi-processes
Asynchronous IO
Serialization & Databases
ContextManager Tools
(much more…)

dir() and help()

- The dir() function displays top-level properties of any object or module



dir() is typically helpful from the interactive shell environment. Use it to identify names of top-level attributes within any object. Help() provides information from the source file (or object's) documentation.

sys Module

- The sys module includes resources for monitoring the Python interpreter environment:

sys.path	list of directories the interpreter uses to locate modules
sys.modules	dictionary of all loaded modules and their file locations
sys.argv	list of command line arguments starting with the file name
sys.builtin_module_names	tuple of strings identifying all pre-compiled modules (no .py file exists for these)
sys.exit(*exit_value***)**	exits the program with an optional exit value
sys.stdin **sys.stdout** **sys.stderr**	file objects that reference the standard streams
sys.executable	where is my python interpreter?
sys.version	gives information about the Python version used
sys.exc_info()	gives type, value, traceback infor for a current exception
sys.getrefcount(obj)	number of references to an object (will always be 1 high)

getrefcount() itself counts as a reference to the object, so this value always reports 1 value too high. Try this:

```
list1 = [1, 2, 3]
list2 = list1
sys.getrefcount(list1)   # 3
```

Using the *sys* Module

```
import atexit
import sys

def cleanup():
    with open('sys_params.txt', 'w', encoding='utf-8') as f:
        print(f'OS: {sys.platform}', file=f)
        print(f'Interpreter location: {sys.executable}', file=f)
        print(f'Module locations: {sys.path}', file=f)

atexit.register(cleanup)

if sys.platform == "win32":
    print('Running Windows...')
    sys.exit()
elif sys.platform == "darwin":
    print('Running OS X')
elif sys.platform == "linux" or sys.platform == "linux2":
    print('Running Linux')
    sys.exit()
```

Registers the function to be automatically called at exit time

Checking Operating System

student_files/ch05_std_lib/01_sys_os_check.py

This example uses the sys.platform variable to illustrate how to perform work based on checking which operating system is used. It also demonstrates how to perform any kind of cleanup actions (if any are desired) when exiting a script by registering a function to be executed when the script ends.

A script may exit at several locations using sys.exit(). To perform any last minute actions, such as saving data to a file, this can be accomplished by writing a function and registering it with the atexit module's register() function.

os and os.path Methods

- *os* provides access to many operating system tools

os.environ	-	a dict of os environment variables
os.getcwd()	-	get the current working directory
os.getenv(env_var, default)	-	returns an environment variable
os.rename(old, new)	-	rename a file or directory
os.remove(filepath)	-	removes a file (errors on directories)
os.rmdir(path)	-	remove a directory
os.chdir(path)	-	change to a new directory
os.mkdir(dir)	-	make a directory
os.makedirs(dirs)	-	make all specified directory levels
os.listdir(path)	-	return a list of files in a dir
os.path	-	submodule with additional attributes
os.path.isfile(file)	-	checks whether item is a file
os.path.isdir(dir)	-	checks whether item is a directory
os.path.exists(path)	-	checks for existence of a file or dir
os.path.basename(path)	-	returns only the last path level
os.path.join(path1, path2, ...)	-	smartly joins paths
os.path.split(path)	-	returns a head and tail where tail is the last part of a path
os.walk(path)	-	walks all files and dirs of a path

The os module is a legacy module in Python but it has also been an important one automation with Python. A useful (website) reference can be found here: https://automatetheboringstuff.com/.

Other helpful third-party tools related to environment automation include:

- Ansible - (https://www.ansible.com/) Automation tool. No Py 3 support, but it's coming!
- Psutil - (https://github.com/giampaolo/psutil/) cross-platform system resources monitoring tool
- Fabric - (http://docs.fabfile.org) - SSH and sys admin support tool
- Paramiko - most common Python SSH tool, Netmiko is a

Using os.walk()

- os.walk() provides a comprehensive way of obtaining information about a directory tree:
  - It returns an iterator, so it must be used iteratively

```
iterator = os.walk(directory)
```

```
for current_dir, subdirs, filenames in os.walk(root_directory):
    # do stuff
```

```
for dirname, subdirs, files in os.walk('..'):
    for filename in files:
        if filename.endswith('.txt') :
            filepath = os.path.join(dirname, filename)
            print(f'{filename:<20}{os.path.getsize(filepath):>8}\
                  {dirname:40}')
```

```
alice.txt          167518  ..\ch01_overview
alice.txt          167518  ..\ch01_overview\solutions
alice.txt          167518  ..\ch02_files_flow_control
...
```

student_files/ch05_std_lib/02_os_walk.py

The example above starts from the root of the student_files directory. It then traverses all directories within student_files returning that directory name (dirname), any subdirectories it has (subdirs), and any files within that directory (files). It then finds .txt files, gets their filesize, and displays the filename, size, and location on the output.

Additional arguments to os.walk() include:

topdown = True. If set to False, it will traverse from the bottom up.

onerror = None. Normally walk() fails silently. Provide onerror=function(err){ } to get error messages.

shutil

- shutil provides high-level methods for copying, moving, removing files and directory trees:

copy(src, dst)	-	copies src file to dst directory, basically Unix cp cmd, copies permissions not date metadata, dst must exist
copy2(src, dst)	-	same as copy but tries to preserve date metadata (uses copystat() internally)
copystat(src, dst)	-	copies permissions & date metadata from src to dst
copyfile(src, dst)	-	copies file with no metadata
copytree(src, dst)	-	copies an entire directory tree. dst directory will be created, permissions/date copied
move(src, dst)	-	moves a src file or directory tree to dst
rmtree(path)	-	removes a directory tree, ignore_errors flag is False

In the method descriptions above, date metadata refers to the creation, last access, and last modification times of the file.

Many methods are similar only varying by how permissions and date statistics are copied.

copystat() is internally called by copy2() and copytree(). It attempts to copy file permissions and date metadata.

Using shutil

```
import shutil

dst = './temp'
copy_dst = './temp/temp2'

for f in os.listdir('.'):
    if f.endswith('.py'):
        shutil.copy(f, dst)


shutil.copytree(dst, copy_dst)
shutil.move(copy_dst, '.')
```

Each file in the ch05 folder that ends with .py will be copied into a subdirectory called temp.

The contents of temp are copied to a new directory called temp2, within temp.

The temp2 directory is moved to the ch05 directory.

student_files/ch05_std_lib/03_shutil.py

This example runs from the current directory, ch05_std_lib. It makes a copy of all files that end with .py and places them in the temp subdirectory within ch05_std_lib. Next, it copies all files from temp into a new directory (that gets created) within temp called temp2. temp2 is moved to the current directory, ch05_std_lib.

Your Turn! Task 5-1

- Use os.walk(), to find the two .jpg images hidden within the student_files
- Make a copy of these files using shutil.copy() and place them in the resources/images folder

Caution: Copying files into from a directory into the same directory causes a SameFileError. This happens when you walk the resources/images directory and try to copy a file from here into here.

To avoid this, either use an if-check or try-except code.

See the starter file for more notes on how to avoid this.

Follow additional instructions within ch05_std_lib/task5_1_starter.py

pathlib.Path

- *pathlib.Path()* is a class that provides an object-oriented approach for working with directories and files

p = Path(file_or_dir)	Creates the Path() object of a file or directory
p_new = p / item	Operator (/) for joining a Path obj and Path or str

```
p_new = working_dir / 'temp.py'
```

p.exists()	Checks if the path item is real *(it doesn't have to be)*
p.is_file(), p.is_dir()	Checks which type of resource it is (file, dir?)
p.parent, p.resolve(),	Returns parent Path obj, returns an absolute path
p.iterdir()	Iterates a directory returning all items (files and dirs)
p.glob(pattern)	Iterates a directory returning items that match pattern
read_text(encoding)	Opens (then closes) a file, reading its contents

The Path() object provides other useful methods, such as: open(), rename(), chmod(), anchor (*drive + root*), drive, cwd(), home(), group(), joinpath(), owner(), parents, mkdir(), rmdir(), and many others.

A good compare-and-contrast to the os module is provided in the docs at https://docs.python.org/3.13/library/pathlib.html#corresponding-tools.

Working with Path()

```
from pathlib import Path

working_dir = Path('.')

if working_dir.exists():
    print(f'Absolute path: {working_dir.resolve()}')


if working_dir.is_dir():
    for p in working_dir.glob('*'):
        print('--> ', p)

for p in working_dir.glob('**/*'):
    print('--> ', p)
```

Create a Path() object wrapping a specified directory or file

Get its absolute path

Get all contents of directory

Get contents of directory and sub-directories (*same as rglob()*)

student_files/ch05_std_lib/04_using_pathlib.py

It takes a little getting used to, but the Path() object provides many methods that can replace the os and os.path methods for working with files and directories.

Your Turn! Task 5-2

- Using pathlib.Path modify the previous task (Task 5-1) by removing the use of the os module and replacing the functionality with Path
- For this solution, handle SameFileError exceptions using the *try-except* approach this time instead of the *if-check* approach used in Task 5-1
- Work from provided task5_2_starter.py file

datetime Module

- The datetime module defines *date*, *time*, and *datetime* classes

date
year
month
day

time
hour
minute
second
microsecond
tzinfo

datetime
year
month
day
hour
minute
second
microsecond
tzinfo

```
from datetime import date, datetime

now1 = datetime.now()
now2 = date.today()

d1 = date(2023, 11, 17)
print(d1.year, d1.month, d1.day)

print(d1.strftime('%d-%b-%Y'))

d2 = date(2023, 11, 27)
print(f'{d2:%Y%m%d}')

print(d2 - d1)
```

```
2023-11-17 18:50:41.669561
2023-11-17
2023 11 17
17-Nov-2023
20231127
10 days, 0:00:00
```

Python 3.12+

student_files/ch05_std_lib/06_using_datetime.py

The formatting characters of the strftime() function can be found in the datetime module's documentation. Python 3.12 added support for the date formatting codes to be used in f-strings directly now.
In addition to what is shown here, dates can be compared:

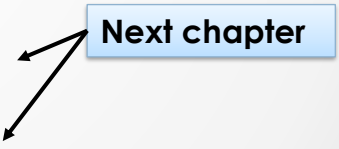
```
date1 > date2 + timedelta(month=1)
```

or

```
d2 = datetime.date(2024, 6, 7)
print(d2 - d)                    # Ans: 476 days 0:00:00  using d from the slide above
```

Parsing Files

- Python can read and write data from numerous source formats
  - CSV files using the **`csv`** module
  - XML using the **`expat`** or **`lxml`** (third-party) tools
  - YAML using the pyyaml (third-party) module
  - JSON data using the **`json`** module
  - HTML parsing using **`html.parser`** module and help from 3rd Party tools such as **`BeautifulSoup`**



csv Module

- The CSV module simplifies working with csv files:

Reading

```
import csv
airports=[]
try:
    with open('../resources/airports.dat', encoding='utf-8') as f:
        try:
            for row in csv.reader(f):
                airports.append(row)
        except csv.Error as err:
            print(f'Error: {err}', file=sys.stderr)
except IOError as err:
    print(err, file=sys.stderr)
```

row represents a list of strings from one line within the file

Writing

```
try:
    with open('first100.dat', 'w', encoding='utf-8') as f:
        try:
            csv.writer(f).writerows(airports[1:101])
        except csv.Error as err:
            print(f'Error: {err}', file=sys.stderr)
except IOError as err:
    print(err, file=sys.stderr)
```

student_files/ch05_std_lib/06_using_csv.py

This example reads from a csv file. It reads one row at a time and automatically parses that row and then puts it into a list (called row here). Each row is appended into an overall list.

Processing XML using ElementTree

- Python provides a built-in parser called: **expat**
  - It supports a number of XML parsing techniques, including *expat*, *sax*, *dom*, *minidom*, and *ElementTree*
- ElementTree is a commonly used API for both creating and parsing XML
- ElementTree API has two primary classes
  - **ElementTree** - XML and document manipulation
  - **Element** - wraps an XML element
- ElementTree.find() and ElementTree.findall() allow for (XPath-like) searching capabilities

While Python supports numerous APIs for parsing and creating XML, ElementTree is one of the most popular options.

When XML is used more frequently, a third-party XML parser called **LXML**, is often a better choice for use due to its improved performance and wider number of features.

Creating XML with ElementTree

- Recall the following data structure from earlier

```
from collections import namedtuple

Contact = namedtuple('Contact', 'first last age email')

records = [
    Contact('John',  'Smith',   43, 'jsbrony@yahoo.com'),
    Contact('Ellen', 'James',   32, 'jamestel@google.com'),
    Contact('Sally', 'Edwards', 36, 'steclone@yahoo.com'),
    Contact('Keith', 'Cramer',  29, 'kcramer@sintech.com')
]
records.sort(key=lambda a: a.age, reverse=True)
```

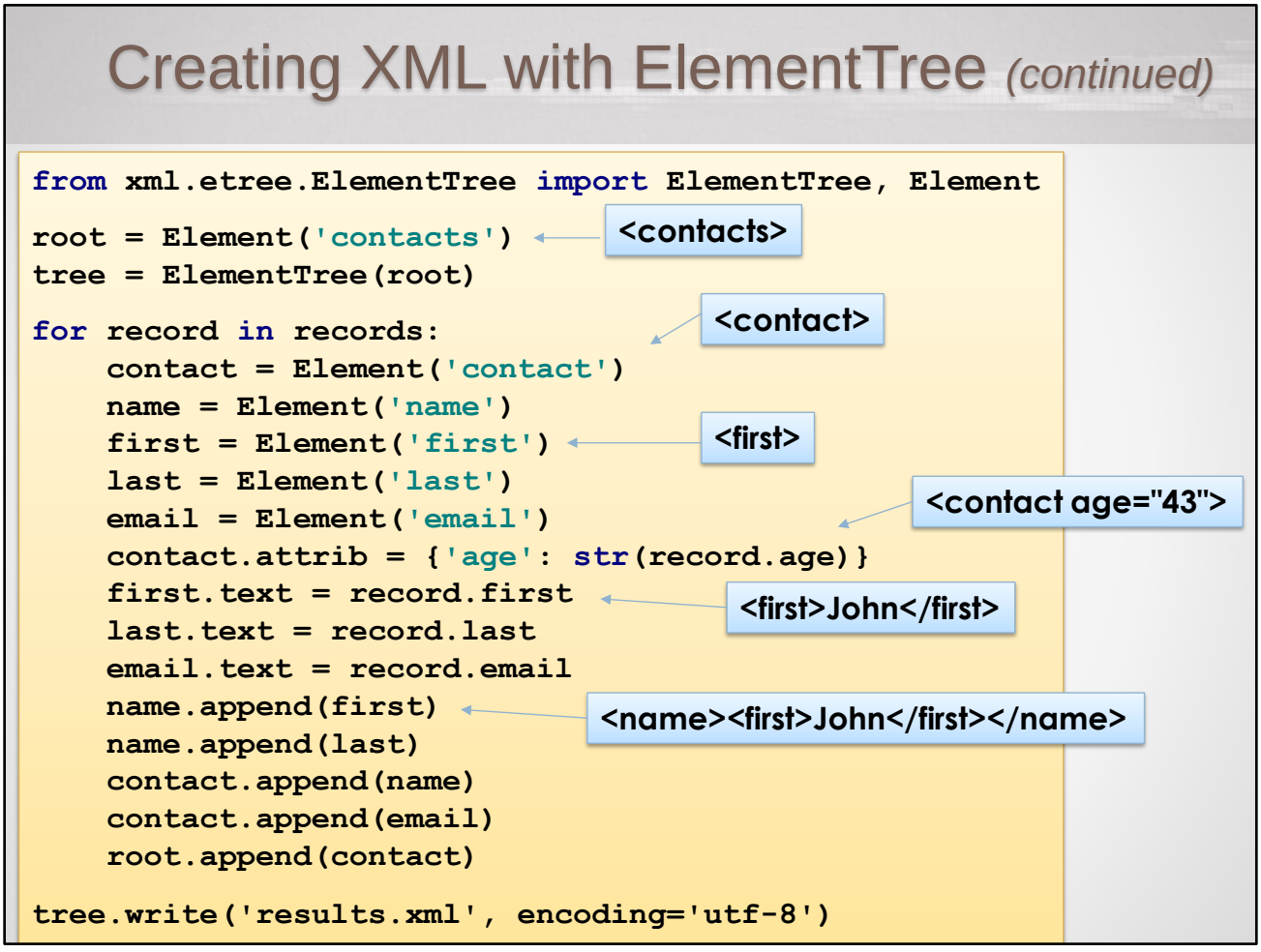
Let's create XML from this data structure...

student_files/ch05_std_lib/07_create_xml.py

The sample data came from an earlier example from chapter 1 called 09_namedtuples.py.

Creating XML with ElementTree *(continued)*

```
from xml.etree.ElementTree import ElementTree, Element

root = Element('contacts')
tree = ElementTree(root)

for record in records:
    contact = Element('contact')
    name = Element('name')
    first = Element('first')
    last = Element('last')
    email = Element('email')
    contact.attrib = {'age': str(record.age)}
    first.text = record.first
    last.text = record.last
    email.text = record.email
    name.append(first)
    name.append(last)
    contact.append(name)
    contact.append(email)
    root.append(contact)

tree.write('results.xml', encoding='utf-8')
```



student_files/ch05_std_lib/07_create_xml.py

This example builds XML from the given data structure on the previous slide. It uses the tree object (ElementTree) to write it out to a file.

XML Results

```
<?xml version="1.0" encoding="utf-8"?>
<contacts>
  <contact age="43">
    <name>
      <first>John</first>
      <last>Smith</last>
    </name>
    <email>jsbrony@yahoo.com</email>
  </contact>
  <contact age="36">
    <name>
      <first>Sally</first>
      <last>Edwards</last>
    </name>
    <email>steclone@yahoo.com</email>
  </contact>
  ...
</contacts>
```

Output from the XML creation code

student_files/ch05_std_lib/07_create_xml.py, results.xml

These are the results from creation of the XML on the previous slide.

Parsing XML with ElementTree

```
from xml.etree.ElementTree import ElementTree

Contact = namedtuple('Contact', 'first last age email')

tree = ElementTree().parse('results.xml')


contacts = []

for contact in tree.iter('contact'):
    first = contact.find('.//first').text
    last = contact.find('.//last').text
    age = contact.get('age')
    email = contact.find('.//email').text
    contacts.append(Contact(first, last, int(age), email))
print(contacts)
```

Parses the entire document

Searches using XPath notation

Iterate over all <contact> elems

Extract the contents of <last>

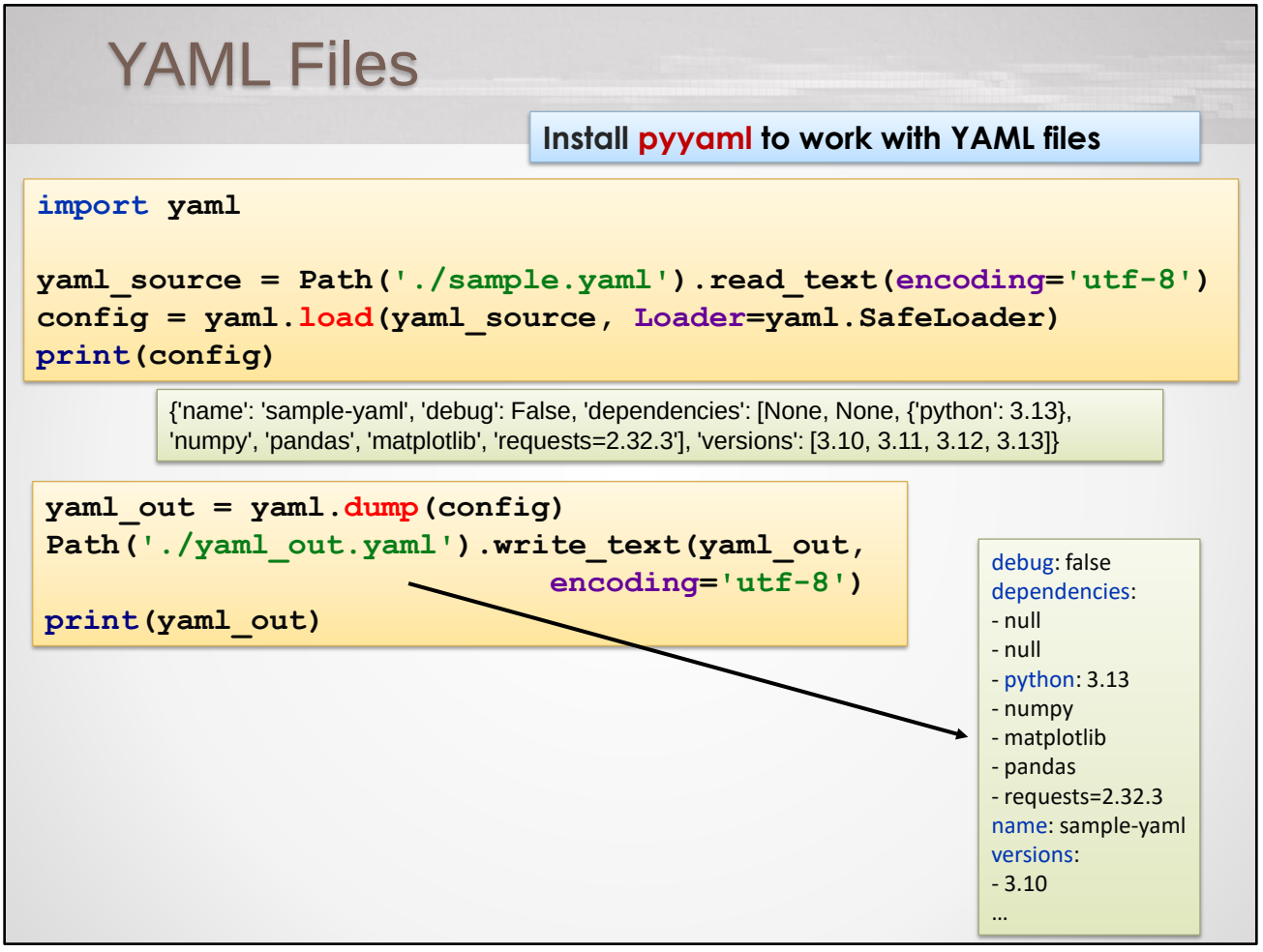
Extract the age attribute

student_files/ch05_std_lib/08_parsing_xml.py

Your Turn! - Task 5-3

- Consume the live XML-based RSS feed
- Capture the title, description, and pubDate *of each item*
- Store the values in a list of named tuples
  - *Note: Internet access is not required for this task; a fallback capability exists to complete it offline*
    (refer to the note at the top of the task5_3_starter.py source file for offline usage)

Follow additional instructions within ch05_std_lib/task5_3_starter.py



student_files/ch05_std_lib/09_using_yaml.py

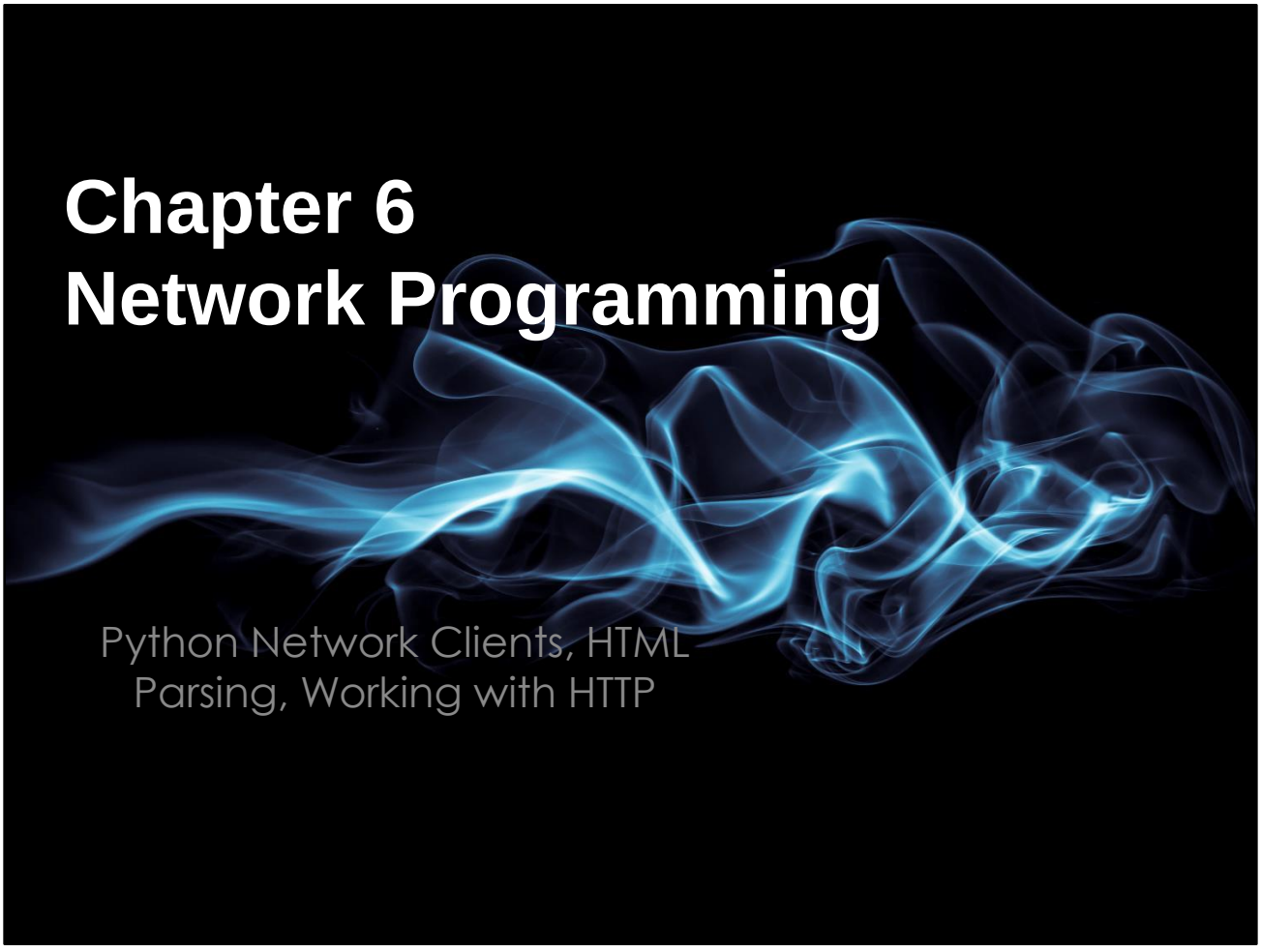
The Python Standard Library doesn't provide a friendly tool for working with YAML files. The common decision is to use the third-party module, pyyaml. It must be installed to use it (pip install pyyaml)

Since YAML contains data values that get converted into Python, some care should be taken. The source of the YAML should be considered. If using an internally created YAML file, then the yaml.FullLoader can be used. However, for untrusted sources, the yaml.SafeLoader should be used.

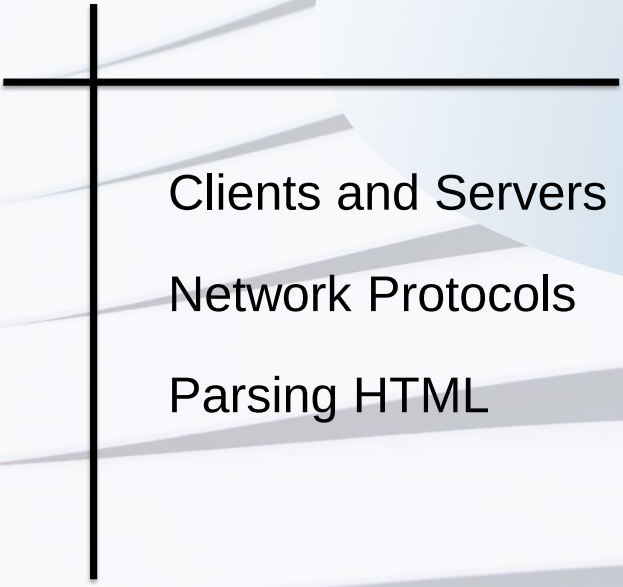
In the example above, *yaml_source* is a string of YAML data. The *yaml.load()* method converts it to a dictionary (config) and *yaml.dump()* converts it from a dict to a string and it then gets saved to the file yaml_out.yaml.

Chapter 5 - Summary

- The *Python Standard Library* contains essential and useful modules for developing Python solutions
  - Explore the online documentation to familiarize yourself with many of these modules
- Different file formats use different techniques to process data
  - Some file types, like YAML, require third-party tools, like pyyaml
- The *ElementTree API* is often a popular choice for working with XML



Chapter 6 - Overview



Clients and Servers

Network Protocols

Parsing HTML

Python and Sockets

- Python provides a low-level interface via the socket module for creating clients and servers

```
import socket
sock = socket.socket(socket_family,
                     socket_type)
```

Usually AF_INET or AF_UNIX

The communications type:
SOCK_STREAM (connection-oriented)
SOCK_DGRAM (connectionless)

Socket Methods

s = socket.socket()	-	creates the socket
s.connect()	-	connect to server socket
s.bind(addr)	-	bind socket to an address
s.listen(backLogQueue)	-	number of queued connections (0 - 5 usually)
s.accept()	-	return is a pair **(conn, address)**, **conn** is a new socket used to send and receive data and **address** the other end of the connection.
s.send(bytes)	-	send data
s.recv(bytes)	-	receive x number of bytes of data
s.close()	-	close connection

The socket family is used to define the addressing format used. For example, an AF_INET family uses the four-number IPv4 address style or domain name style + a port. (i.e., *host, port*). AF_INET is the most common value for the socket family however other values include AF_UNIX (bound to a system node), AF_INET6 (using IPv6 format), AF_NETLINK, AF_TIPC, AF_BLUETOOTH, AF_PACKET, AF_CAN, etc.

Simple Socket Requests

- Sockets provide low level communication capabilities…

```
host = 'docs.python.org'
port = 80
request = b'GET / HTTP/1.1\r\nUser-Agent: Not Mozilla\r\n
                        Host: docs.python.org\r\n\r\n'

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ip_address = socket.gethostbyname(host)
s.connect((ip_address, port))
s.sendall(request)
resp = s.recv(4096)
s.close()
print(resp)
```

student_files/ch06_network_prog/01_client.py

For brevity, exception handling is not shown (see the source file). The example shows the use of a socket on a client making a request to a server. It makes an HTTP request to a host's port 80.

This example is limited, however, as most hosts will perform a redirect to their secure port (443). To support this redirect, a secure socket would be needed to continue the communications (see next slide).

Higher-Level Interfaces: *urllib*

- The urllib package provides high-level requests

```
from urllib.request import urlopen
from urllib.parse import urlparse

url1 = 'https://httpbin.org/html'
url2 = 'https://httpbin.org/encoding/utf8'

with urlopen(url1) as f:
    results = f.read().decode('utf-8')
    print(results)


# shortened version...(but doesn't close)
print(urlopen(url2).read().decode('utf-8'))


result = urlparse('https://docs.python.org/3/library/index.html')
print(result)
print(result.netloc)
```

with control closes connection

Eventually times out but may leave resources stranded in memory

docs.python.org

It Task1-3 the fast way!

student_files/ch06_network_prog/02_urllib.py

Refer to the student files example to see the output of the first two examples above.

The last result above is a ParseResult object as follows:

ParseResult(scheme='https', netloc='docs.python.org',
path='/3/library/index.html', params='', query='', fragment='')

urlopen() Error Handling and *with*

```
from urllib.request import urlopen
from urllib.error import URLError, HTTPError

url200 = 'https://httpbin.org'
url404 = 'https://httpbin.org/foo'
url403 = 'https://httpbin.org/status/403'
urlerr = 'https:/httpbin.org'

for url in url200, url404, url403, urlerr:
    try:
        with urlopen(url) as f_url:
            results = f_url.read().decode('utf-8')
            print(results)
    except HTTPError as err:
            if err.code == 404:
                print('Page not found.  Bad URL.', file=sys.stderr)
            elif err.code == 403:
                print('Access denied.', file=sys.stderr)
            else:
                print('An HTTP error occurred.', file=sys.stderr)
    except URLError as err:
        print(f'Error: {err}', file=sys.stderr)
```

Can we write the error handling another way?

URLError is the parent to HTTPError so it goes last

student_files/ch06_network_prog/03_with_urlopen.py

Just like sockets, urlopen() can be used in a *with* control in Python 3. It will automatically close the connection (socket) for us.

Using Match-Case Control

- Python supports a "switch"-style control *(added in Python 3.10)*
  - Only one case can be evaluated, there is no "fall-through"

```
url500 = 'https://httpbin.org/status/500'
url404 = 'https://httpbin.org/foo'
url403 = 'https://httpbin.org/status/403'
```

```
for url in url500, url404, url403:
    try:
        with urlopen(url) as f:
            results = f.read().decode('utf-8')
            print(results)
    except HTTPError as err:
            match err.code:
                case 404:
                    print('Page not found.  Bad URL.', file=sys.stderr)
                case 403:
                    print('Access denied.', file=sys.stderr)
                case _:
                    print('An HTTP error occurred.', file=sys.stderr)
    except URLError as err:
        print(f'Error: {err}', file=sys.stderr)
```

student_files/ch06_network_prog/04_match_case.py

The match-case control is somewhat controversial as some feel it is unnecessarily adding features to the language that weren't necessarily needed. However, it was written to be very powerful and will likely find more and more use cases over time.

Our revised error handler is using the Python 3.10 match-case control. The match-case control can be used to parse command-line arguments, process log file levels, or perform pattern matching to extract parts of data structures.

Unlike the "switch" version from other languages, once a case is matched, no other cases will then be matched again and there is no "fall through," meaning because there is no break, the next case *does not* get executed.

requests

pip install *requests*

- requests is a third-party (preferred) module that simplifies making HTTP requests

```
import requests

r = requests.get("https://httpbin.org/json")
```

```
print(r.text)
print(r.json())

print(r.url)
print(r.status_code)
print(r.headers)
print(r.request.headers)
```

```
r = requests.post("http://someURL", data = {'key': 'value'})
r = requests.put("http://someURL", data = {'key': 'value'})
r = requests.delete("http://someURL")
r = requests.head("http://someURL")
r = requests.options("http://someURL")
```

student_files/ch06_network_prog/06_requests.py

The *requests* module is a popular third-party module used to simplify HTTP requests.

html.parser

```
from html.parser import HTMLParser
import requests

class PageParser(HTMLParser):
    def __init__(self):
        super().__init__()
        self.marker = False
        self._info = []

    def handle_starttag(self, tag, attribs):
        if tag == 'h1' or tag == 'title':
            self.marker = True

    def handle_data(self, data):
        if self.marker:
            self.marker = False
            self._info.append(data)

page_parser = PageParser()
page_parser.feed(requests.get('https://www.cisco.com').text)
print(f'<title> and <h1> elems:\n{ page_parser.info }')
```

Records contents of the <title> and <h1> tags

student_files/ch06_network_prog/07_html_parse.py

This example uses the HTMLParser class of the html.parser module (formerly htmllib) to parse a specified web page. It operates at a fairly low-level, using a SAX-style parsing of capturing events by calling the handle_starttag() and handle_data methods().

Beautiful Soup

pip install *beautifulsoup4*

- Beautiful Soup is a popular third-party Python add-on for screen scraping and HTML parsing
  - Easier to use than the html.parser module

```
from bs4 import BeautifulSoup
import requests

page = requests.get('https://www.python.org').text
soup = BeautifulSoup(page, 'html.parser')

print(soup.title)
all_h2 = soup.find_all('h2')
print(len(all_h2))
print([elem.text for elem in all_h2])
```

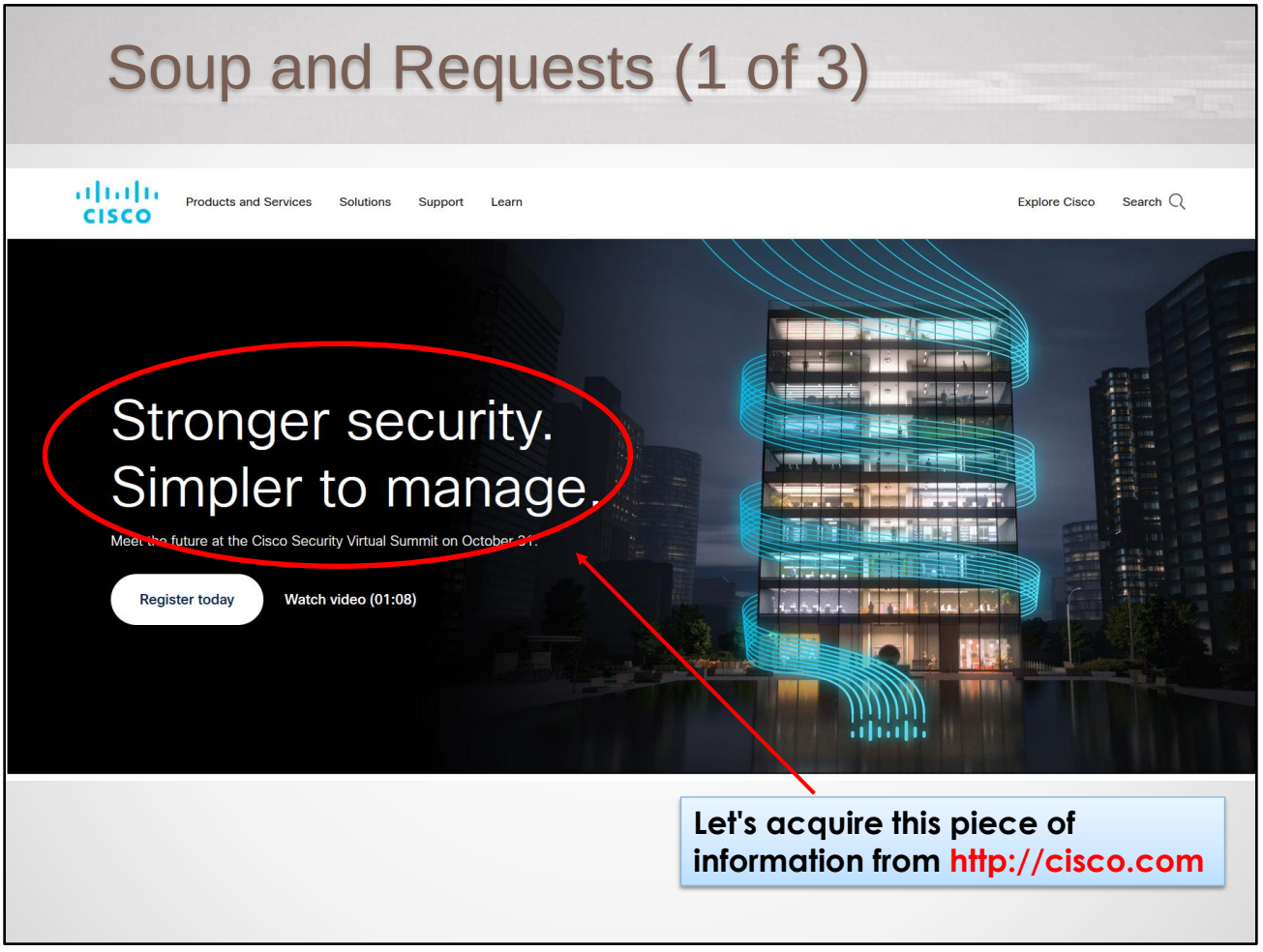
```
10
```

```
['Get Started', 'Download', 'Docs', 'Jobs', 'Latest News', 'Upcoming Events', 'Success Stories', 'Use Python for…', '\n>>> Python Enhancement Proposals (PEPs): The future of Python is discussed here.\n RSS\n', '\n>>> Python Software Foundation\n']
```

student_files/ch06_network_prog/08_soup.py

Home page: https://www.crummy.com/software/BeautifulSoup/
Docs can be found at: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Take special care: the install package name (beautifulsoup4), the module import name (bs4) and the main class used (BeautifulSoup) are all different names!



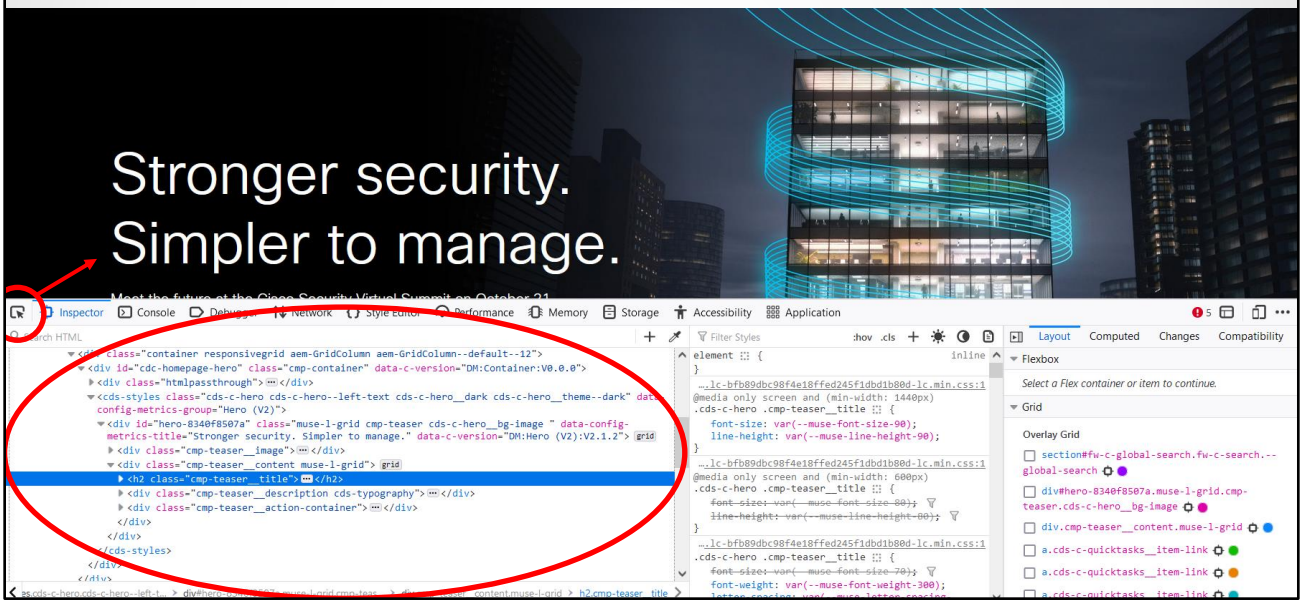
student_files/ch06_network_prog/09_requests_and_soup.py

As a demonstration of retrieving content from an HTTP server, we will obtain the header content from Cisco's home page.

Note: this page is live and dynamic and will not look the same as the screenshot shown above. Instead, the main headline should be the one retrieved if the page is different.

Soup and Requests (2 of 3)

- Inspecting the HTML in a browser's developer tools *(usually opened with CTRL-Shift-I or Cmd-Shift-I, or F12)* reveals the HTML structure



student_files/ch06_network_prog/09_requests_and_soup.py

By opening the developer tools within the browser (any major browser), the picker tool is selected and used to zero-in on the desired HTML element. This helps us identify the location within the HTML document that contains the relevant content.

Soup and Requests (3 of 3)

- First, we'll use *requests* and then pass the results on to *BeautifulSoup*

```
from bs4 import BeautifulSoup
import requests

content = requests.get('https://cisco.com').text

soup = BeautifulSoup(content, 'html.parser')
print(soup.title.text)


print('h2\'s on the page:')
for elem in soup.find_all('h2'):
    print(elem.text)

selector = 'h2.cmp-teaser__title'
headline = soup.select(selector)
if headline:
    print(headline[0].text)
```

The HTML page as a string

Networking, Cloud, and Cybersecurity Solutions - Cisco

Displays several H2 tags (live results will vary)

Displays headline (live results will vary)

student_files/ch06_network_prog/09_requests_and_soup.py

Finally, we'll use requests and BeautifulSoup to acquire our content. The soup.select() statement can obtain content using a CSS selector. Results are returned in a list, so the first one (and only one) will be used.

json Module

- Python provides a json module for converting objects to (and from) the JSON data format

```
import json
```

- Use json.dumps() to convert from dict to JSON

```
import json
task = {'task': 'run 5 miles', 'goal': 40}
print(json.dumps(task, indent=4))
```

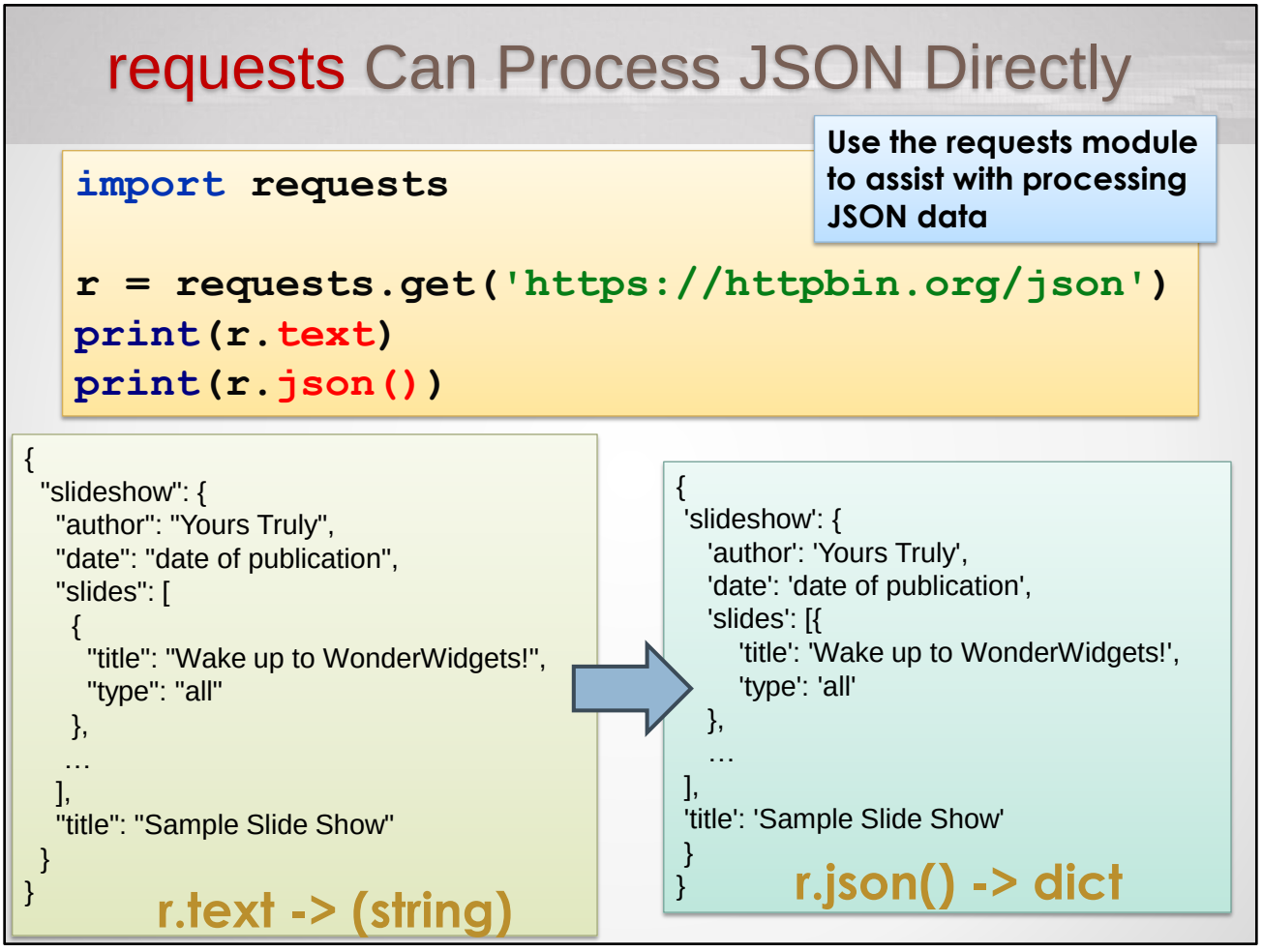
```
{
    "task": "run 5 miles",
    "goal": 40
}
```

- Use json.loads() to convert from JSON string to a dict

```
new_dict = json.loads('{
"first": "John","last": "Smith",
"age": 43,"email": "jsbrony@yahoo.com"}')
print(new_dict)
```

```
{
    "first": "John",
    "last": "Smith",
    "age": 43,
    "email": "jsbrony@..."
}
```

student_files/ch06_network_prog/10_json.py

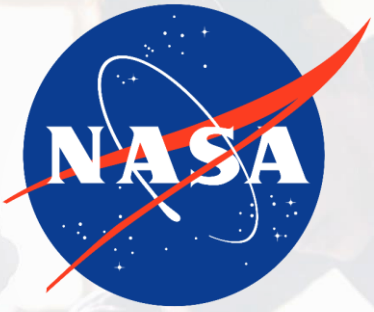


student_files/ch06_network_prog/11_json_requests.py

Despite its misleading name, the .json() method of the requests library returns a dictionary.

Your Turn! Task 6-1 - *Overview (1 of 4)*

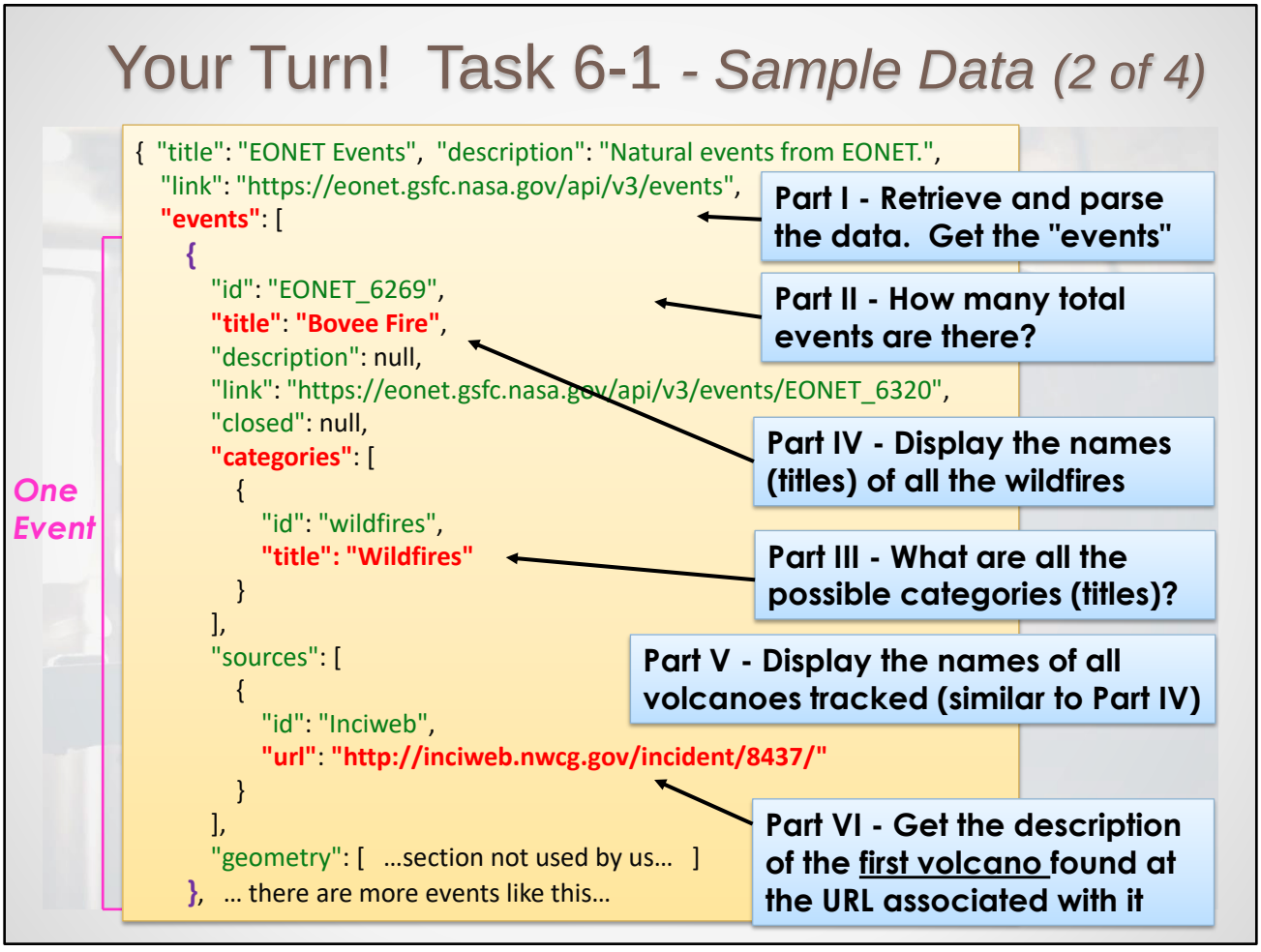
- Use the *NASA Earth Observatory Natural Event Tracking* API to retrieve information on ***volcanoes*** and ***fires***



Follow specific instructions within ch06_network_prog/**task6_1_starter.py** or within the back of the student manual

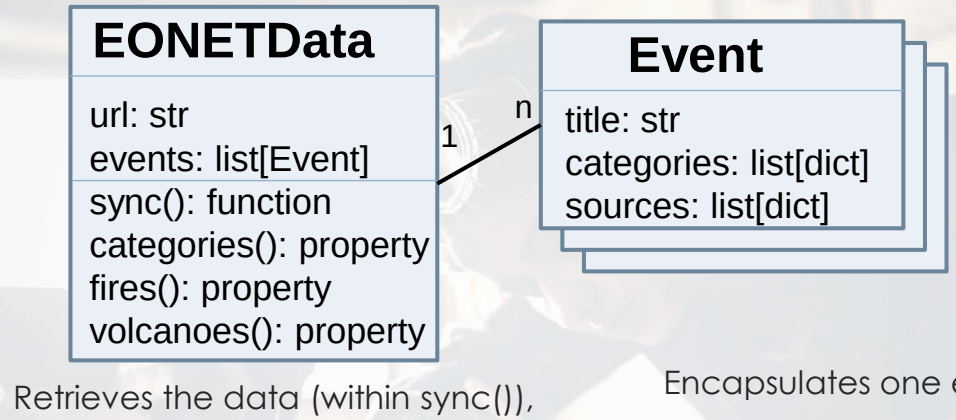
Note: if no internet access is available, uncomment the import mocklab statement

Please note that because this is live data, the structure of the data does occasionally change.



The data example above shows a "Wildfires" event type, however, a "Volcanoes" event type has the same structure.

Your Turn! Task 6-1 - *Classes (2 of 4)*



Retrieves the data (within sync()), holds the list of events tracked, provides access to categories, fires and volcanoes

Encapsulates one event

Your Turn! Task 6-1 - *The Parts (4 of 4)*

- For this task complete these *6 parts*:

Part I

- Create an Event class. Complete the EONETData class to have it retrieve the JSON data from the following URL: https://eonet.gsfc.nasa.gov/api/v3/events

  Build properties to track the *categories, fires* and *volcanoes*

Part II

- Determine how many total events are being tracked (hint: use len())

Part III

- Determine all the different types of *category titles* being tracked *(see sample JSON on previous slide)*

Part IV

- Print the titles of all the *Wildfires* types

Part V

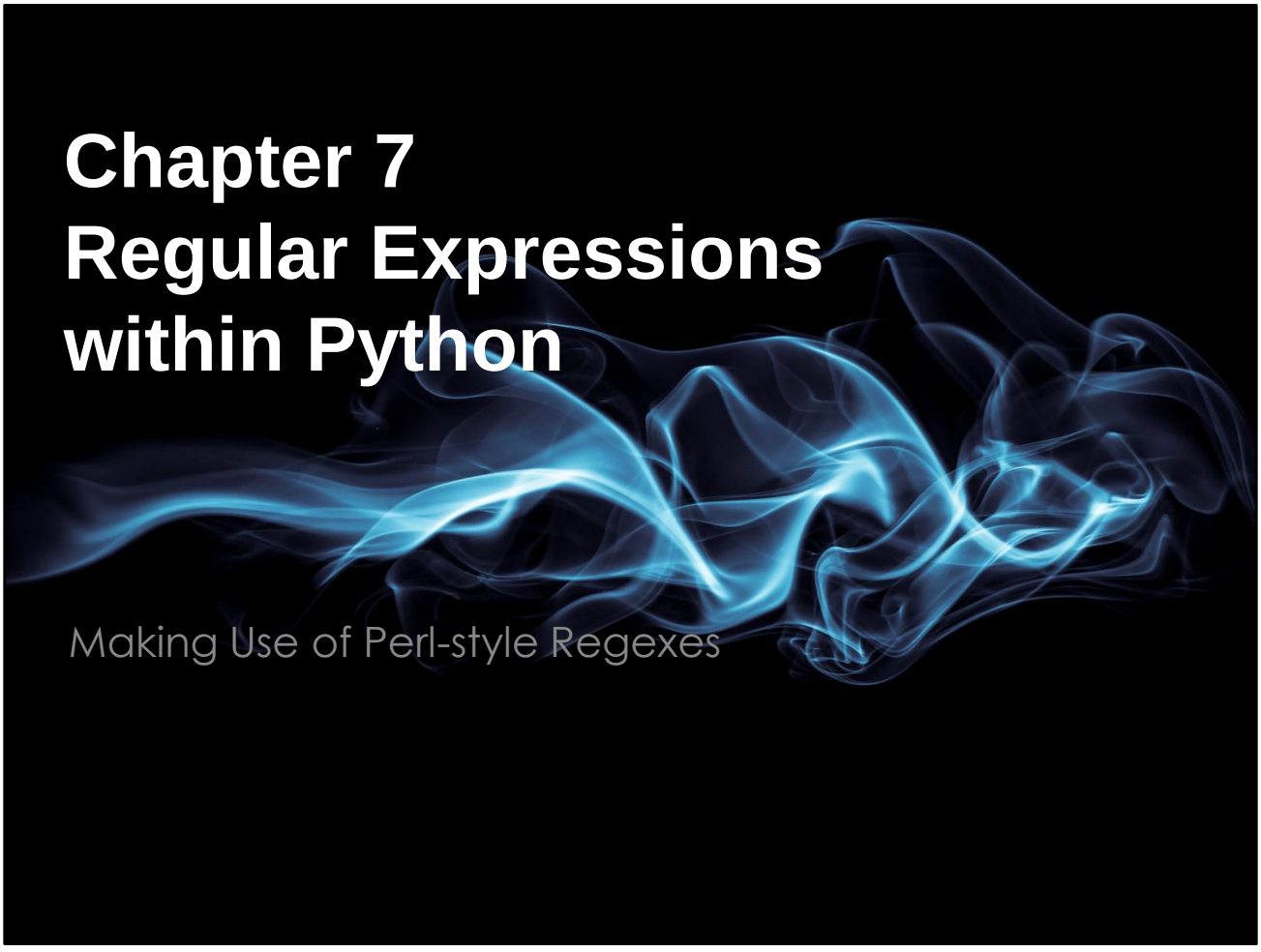
- Display the titles of all the *Volcanoes*

PartVI

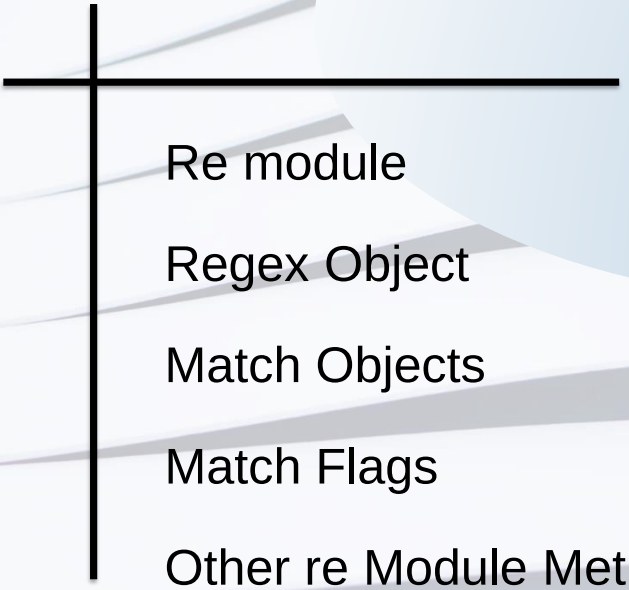
- Display the description for the first volcano tracked

Chapter 6 - Summary

- Python offers several levels of resources for working with network activities
  - Low-level: sockets
  - Mid-level: protocol-specific modules (e.g., ftplib)
  - High-level: urllib.request
- Third-party tools such as requests and BeautifulSoup provide improvements to tools provided by the standard library
- Parse JSON data using the json module
  - Create encoders/decoders to handle fields that do not convert properly



Chapter 7 - Overview

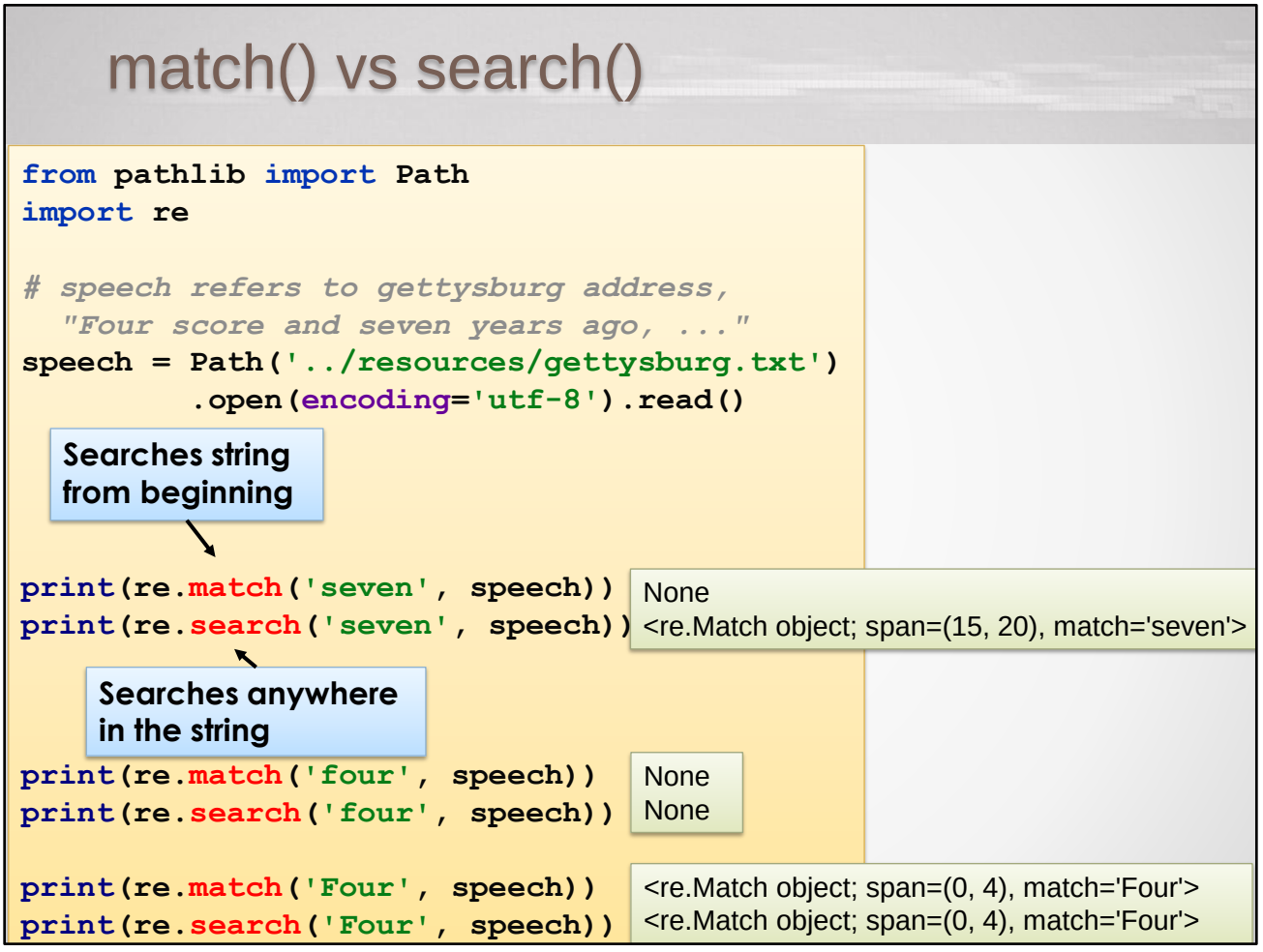


- Re module
- Regex Object
- Match Objects
- Match Flags
- Other re Module Methods

re Module Method Summary

- Python supports Perl-style regex's via the `re` module
- Module-level methods include

match(pattern, string, flags)	-	from the beginning, return Match object if a match exists, None otherwise
search(pattern, string, flags)	-	search entire string for match, return Match object if exists, None otherwise
findall(pattern, string, flags)	-	list of matches of patterns within string
finditer(pattern, string, flags)	-	iterator of matches of patterns in string
fullmatch(pattern, string, flags)	-	apply pattern to full string, Match object or None returned
split(pattern, string, maxsplit, flags)		break string up by regex pattern
sub(pattern, repl, string, count, flags)		find match, replace repl with it. Return new string.



student_files/ch07_regexes/01_matching.py

In this example, both the match() and search() methods are explored. The match() will only consider a match if it occurs at the beginning of the string, while search() considers any location within the string valid.

The results indicate that a match is not found when using re.match() with the pattern of "seven." That's because "seven" does not occur at the beginning of the string. It does, however, occur within the speech, so a MatchObject is returned for the first re.search() call.

The second pair of calls both do not find "four" because in the speech, "Four" is capitalized.

The last set of calls each find "Four" because it is at the beginning (and within) the speech.

Using Raw Strings

- To avoid confusion by having to escape characters within a regex string, use raw strings:

```
matchobj = re.match('\\d{5}', '12345')

matchobj = re.match(r'\d{5}', '12345')
```

With raw strings, backslashes are not treated as special character

Raw strings should be used when processing complex regexes that use many of the special characters found on the next slide.

Common Regexes

Symbol	Meaning
^	from the start
$	to the end
.	any character
\s	whitespace
\S	non-whitespace
\d	digit
\D	non-digit
\w	alphanumeric character
\W	non-alphanumeric character
\b	word boundary
\B	non-word boundary

Symbol	Meaning
*	0 or more
+	1 or more
?	0 or 1
{n}	exactly n
{5,8}	5 to 8
{5,}	5 or more
{,8}	Up to 8
(1|2|3)	1 or 2 or 3
[adrn]	a or d or r or n
[a-f]	one of a thru f
[^def]	not d or e or f
[a-zA-Z]	one of any letter

Python supports most common PERL-style regexes. Here are a few of the common ones. Others are supported. For full listing, visit:

https://docs.python.org/3/library/re.html

Python 3.3+ supports unicode chars in regexes by using \u in a raw string.

Match Objects

- A Match object is returned from either match() or search()

```
matchobj = re.search('seven', speech)
if matchobj:
    print(f'seven found at pos: {matchobj.start()}')
```

- Match object methods:

start()	-	index of the start of the match
end()	-	index of the end of the match
span()	-	both values (start, end)
groups()		returns a tuple of all sub-groups (parenthesized expressions)
group(n)		specified sub-group, zero is the whole match

student_files/ch07_regexes/01_matching.py

The Match object is the returned object instance of an re.search() or re.match() call. Match objects can be placed into a conditional and tested directly, or if additional info is needed, you can invoke one of the Match object instance methods listed above.

Groupings

- When a match occurs, matchobj.groups() will return a tuple of the whole match and any subgroups
  - Use matchobj.group(n) to obtain the subgroup

```
matchobj = re.search(r'(\w+) (\w+) (\w+)',
                     'Four score and seven years ago')

print(matchobj.groups())       # ('Four', 'score', 'and')
print(matchobj.group(0))       # Four score and
print(matchobj.group(1))       # Four
print(matchobj.group(2))       # score
print(matchobj.group(3))       # and
```

student_files/ch07_regexes/01_matching.py

matchobj.group(0) will return the entire match. To get the subgroups, use matchobj.groups() to find the length and query matchobj.group(n) each time to extract each individual grouping value.

Using findall()

- The findall() method allows for finding multiple occurrences of a regex
  - Returns a list of strings that match

```
str_matches = re.findall(r'\w+', 'Four score and seven years ago')

print(f'How many words: {len(str_matches)}')
```

How many words: 6

```
print(str_matches)
```

['Four', 'score', 'and', 'seven', 'years', 'ago']

student_files/ch07_regexes/01_matching.py

The main difference between findall() versus search() is that it will return a list of strings that match. search() returns a single Match object that stops after finding the first occurrence.

Matching Flags

- Flags can be set to tailor aspects of the search

re.IGNORECASE	**- case insensitive matches**
re.VERBOSE	**- use more verbose-style regular expressions**
re.DOTALL	**- dot (.) can match any char including newlines**
re.MULTILINE	**- matches at the beginning of each line are allowed with match()**

Verbose Flag

```
pattern = r'''
(\(?\d{3}\)?)?    # optional area code, parentheses optional
[-\s.]?           # opt. separator: dash, space, or period
\d{3}             # 3-digit prefix
[-\s.]            # separator: dash, space, or period
\d{4}             # final 4-digits
'''

phones = [
    '123-456-7890',
    '800 555 4400',
    '(123) 456-7890',
    '123.456.7890',
    '123-4567',
    'reallywrong',
    '1234-456-7890'
]

valid = [ph for ph in phones if re.match(pattern, ph, re.VERBOSE)]
print(f'Valid phones: {valid}')
```

```
Valid phones: [
  '123-456-7890',
  '800 555 4400',
  '(123) 456-7890',
  '123.456.7890',
  '123-4567'
]
```

student_files/ch07_regexes/02_verbose.py

String Manipulation

- Two methods can be used to manipulate strings after a search has been performed

```
newstr = re.sub(pattern, replacement, sourcestring)
```

Replaces first match in sourcestring with 'replacement'

```
new_str = re.sub('seven', 'eight',
                 'Four score and seven years ago')
print(new_str)
```

Four score and eight years ago

```
re.split(pattern, sourcestring)
```

Breaks a string into a list based on a specified pattern

```
print(re.split(r'\d+', 'hello1234567890world'))
```

['hello', 'world']

Compiling for Efficiency

- Use the reg = re.compile(regex) method if a regex will be utilized repeatedly

```
pattern_obj = re.compile(pattern, re.VERBOSE)

valid = [ph for ph in phones if pattern_obj.match(ph)]
```

This is a compiled expression, so repeated use of it can save regex compilation time

student_files/ch07_regexes/02_verbose.py

Your Turn! Task 7-1

- Revisit the *alice.txt* exercise (Task 1-5) where the top 100 occurring 5-character words were found
  - For this task *use regular expressions* to remove punctuation (period, question mark, exclamation mark, double-quotation mark, colon, comma)
  - *Also convert words (keys) to lower case*

Follow additional instructions within ch07_regexes/task7_1_starter.py

Chapter 7 - Summary

- Use the compile() method of the re module for efficiency when the regex will be used repeatedly
- Supply *raw strings* and use verbose flag to make regexes more readable and maintainable
- Use the module-level search methods for simplicity
- Iterate over groups returned from matches to obtain the sub-groups

What Did We Learn?

- Data Types
- Operations of Sequences
- Slicing
- String and date formatting
- Comments and Docstrings
- Importing and managing modules
- Executing scripts various ways
- Networking Modules (urllib, requests)
- Control Structures
- Various iteration techniques
- Sequences: Lists, Tuples, Strings
- List comprehensions
- Sets, Dictionaries
- Sorting Dictionaries
- Creating Functions
- Variable Scoping
- Using default parameters
- Multiple positional parameters
- Multiple keyword parameters
- Exception Handling
- Creating classes
- Constructors
- Namedtuples
- Inheritance Basics
- Class variables
- File I/O
- Std Lib Modules: sys, os, time datetime, csv, re, pathlib, shutil
- Handling XML and JSON

Intensive Intermediate Python
What's in Part II?

- Review of Fundamentals
- Working with Databases
- Python DB API 2.0
- SQLAlchemy
- Class & Static Methods
- More with Magic Methods
- super()
- Inheritance
- Multiple Inheritance
- subprocesses
- Threads
- Global Interpreter Lock
- Locks, Queues
- collections Module
- itertools Module and Iterators
- Multi-processing
- argparse
- Closures
- Decorators
- Functional Programming
- Generators, Generator Expressions
- Dict & Set Comprehensions
- logging
- functools
- WSGI and Web Frameworks
- RESTful Frameworks
- Flask

Evaluations

- Please take the time to fill out an evaluation
- All evaluations are read and considered

Questions

Questions?

Intensive Introduction to Python

Exercise Workbook

Task 1-1
Python Environment Setup and Test

Overview

This exercise will help ensure that your working environment is properly set up. If you have been provided with a pre-configured machine or virtual desktop, you should be able to skip this task, however, it may be useful to review it to better understand what has already been done for you.

Install Python

If you have not already done so, install Python by visiting **`https://www.python.org/downloads/`**.

Click the link to download Python 3.x.

Click the link (circled in the above diagram) to download the appropriate version for your platform.

Run the installer and select the custom install option.

Install Python ensuring you ***Add Python to the Path*** (watch for this option during the installation process). Also, if your permissions will allow it, select the ***Install for all users***.

Afterwards, **open a console or terminal** window and type:

```
python -V
```

If the command is not recognized or the wrong python version is displayed, you will need to modify your PATH environment variable to include the <PYTHON_HOME> directory. Setting the PATH varies on operating systems, so ask for help if assistance is needed.

Note for Macs

*Use the command: **python3.13** after installing to invoke the 3.13 version of Python instead of the installed default version. Substitute 3.12, 3.11, 3.10, etc., for your installed version. **python3** without the minor version often works as well.*

One of these commands should work for you. If not, you will need to check your PATH environment variable and double-check the location that Python was installed.

Jot down or remember the correct python command that works for you as it will be used in this course!

In this course, replace any Python command-line references (e.g., python, python3, python3.13, etc.) with the syntax that you just determined works for you!

Test the Python Shell

At a terminal/command prompt, type **`python`** `(or python3, python3.13, python3.x, etc.)`. Once the Python interactive shell starts, you should see **>>>** as a prompt.

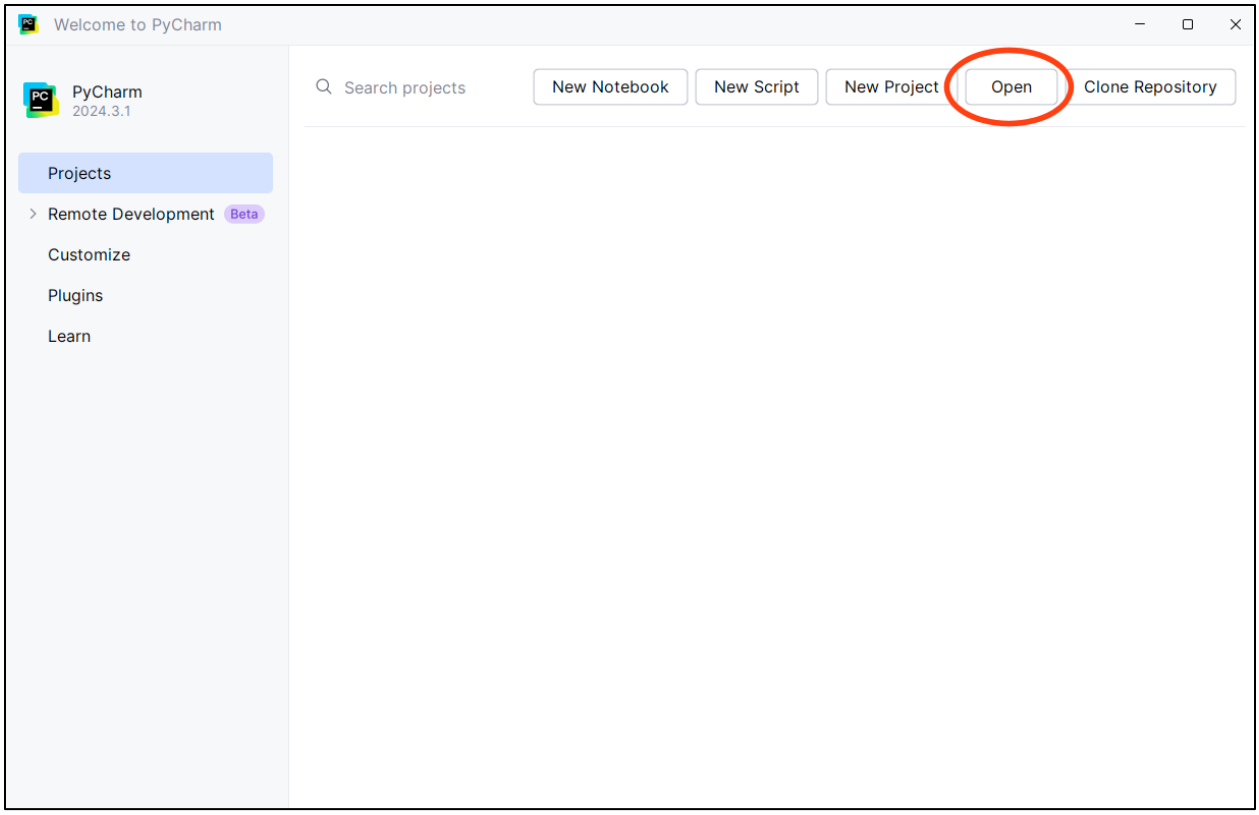
Type the code shown below followed by the *Enter* key after each line:

```
str1 = 'hello'
str2 = 'world'
print(str1, str2)
```

Launch PyCharm, Set Up Student Files

PyCharm Community Edition is a free Python IDE created by JetBrains. After launching PyCharm, a "Welcome to PyCharm" dialog will appear. Select "Open..." and browse to the **student_files** directory where your student files are.

Note: if instead of the "Welcome" dialog your PyCharm launches directly into the IDE, you may skip the rest of this step.



Once PyCharm starts, it may prompt you to create a virtual environment. Accept the defaults. This process will include installing additional dependencies defined in a file called *requirements.txt*.

If PyCharm does not prompt you automatically to create the virtual environment, you can do it yourself, manually, by following the instructions at the top of the file called requirements.txt found within your student files.

Run Your First Script

Open task1_1_starter.py from your `student_files/ch01_overview` folder.

Right-click in the source code and select Run `task1_1_starter.py`.

You should see results from running this script. If not, your instructor may have to help you set the interpreter.

NOTE: In this workbook, occasionally lines of code will be written like this:

```
people_filepath =
        os.path.join(working_dir, people_filename)
```

These long lines should be written all on one line if possible.

This concludes Task 1-1.

Task 1-2
Working with Strings and Controls

Overview

This exercise is intended to provide practice with Python strings and control structures. You will work with a string URL, parse it, and extract ONLY the domain name portion of it. You will work from the *task1_2_starter.py* file found in the *ch01_overview* folder of the student files for this exercise.

Work with a URL

Uncomment one of the provided URLs.

```
prefixes = ['http://', 'https://']
suffixes = [':', '/', '?']

url1 = 'https://docs.python.org/3/'
# url2 = 'https://www.google.com/search?q=python'
# url3 = 'http://localhost:8005/contact/501'
```

Remove the Protocol Prefix

Check to see if the URL begins with "http://" or "https://" and then remove it from the URL. Do this using *startswith()* and slicing.

Iterate over the provided prefixes checking to see if the string starts with that prefix.

```
prefixes = ['http://', 'https://']
suffixes = [':', '/', '?']

url1 = 'https://docs.python.org/3/'
# url2 = 'https://www.google.com/search?q=python'
# url3 = 'http://localhost:8005/contact/501'

for prefix in prefixes:
    if url1.startswith(prefix):
        domain = url1[len(prefix):]
```

Remove the URL Suffix

Repeat the above step this time removing any content *after* the domain name. A list of suffixes was provided, use this list invoking the string class's find() method. If find() returns -1 then the suffix is not in the string. Otherwise, find() will return the position of the suffix character within the string. Take the slice from the beginning of the string to this position.

```
prefixes = ['http://', 'https://']
suffixes = [':', '/', '?']

url1 = 'https://docs.python.org/3/'
# url2 = 'https://www.google.com/search?q=python'
# url3 = 'http://localhost:8005/contact/501'

for prefix in prefixes:
    if url1.startswith(prefix):
        domain = url1[len(prefix):]
```

```
for suffix in suffixes:
    pos = domain.find(suffix)
    if pos != -1:
        domain = domain[:pos]
```

Print the String

Print the remaining string after the prefix and suffix have been stripped off. Your final result will look like the following:

```
prefixes = ['http://', 'https://']
suffixes = [':', '/', '?']

url1 = 'https://docs.python.org/3/'
# url2 = 'https://www.google.com/search?q=python'
# url3 = 'http://localhost:8005/contact/501'

for prefix in prefixes:
    if url1.startswith(prefix):
        domain = url1[len(prefix):]

for suffix in suffixes:
    pos = domain.find(suffix)
    if pos != -1:
        domain = domain[:pos]

print(domain)
```

Here's the "advanced" version that includes working with all three URLs iteratively (be sure to uncomment all three of them at once):

```
urls = [url1, url2, url3]

for url in urls:
    domain = url
    for prefix in prefixes:
        if url.startswith(prefix):
            domain = url[len(prefix):]

    for suffix in suffixes:
        pos = domain.find(suffix)
        if pos != -1:
            domain = domain[:pos]

    print(domain)
```

Task 1-3
Control Structures and Files

Overview

This exercise provides the ability to search files for a specified string expression much like a grep command. The solution will search either a directory of files or a list of file names using the syntax shown below. Note: this solution doesn't read binary files (non-text files).

python task1_3_starter.py wordexpression directory

Create the List of Files

Obtain the command-line arguments (wordexpression and directory). Use os.listdir() to return the items in the directory.

```
args = sys.argv

if len(args) != 3:
    print('Insufficient arguments provided. Syntax:')
    print('python task1_3.py wordexpression directory')
    sys.exit(42)

word_expression = args[1]
directory = args[2]
file_list = []

# put your solution here
dir_contents = os.listdir(directory)
for entry in dir_contents:
    filename = os.path.join(directory, entry)
```

```
    if os.path.isfile(filename):
        file_list.append(filename)
```

Iterate Over the List of Files, Open A File

You now have a list of files (called file_list). At the end of the provided starter code, you can iterate over the newly created file_list. Take one file from the file_list and begin reading from it using another for loop.

```
dir_contents = os.listdir(directory)

for entry in dir_contents:
    filename = os.path.join(directory, entry)
    if os.path.isfile(filename):
        file_list.append(filename)

for filename in file_list:
    for line in open(filename, encoding='utf-8'):
```

Read a Line, Check It for the Expression

Once the file is open, read one line at a time from the file. Use the str class find() method to search for the expression within the line. Keep track of line numbers with a variable.

```
dir_contents = os.listdir(directory)

for entry in dir_contents:
    filename = os.path.join(directory, entry)
    if os.path.isfile(filename):
        file_list.append(filename)
```

```
for filename in file_list:
    line_count = 0
    for line in open(filename, encoding='utf-8'):
        line_count += 1
        if line.find(wordexpression) != -1:
            # see next step
```

Display the Line Number if There is a Match

If there is a match (determined by the last line in the previous step, then print the line number, filename, and the word expression:

```
print(f'File: {os.path.basename(filename)},
               Line: {line_count}, ({wordexpression})')
```

Testing using Command-Line Arguments

Test your solution either from a command/terminal window, or from an IDE:

1. Using a command/terminal window: **cd** to the ch01_overview folder and type:

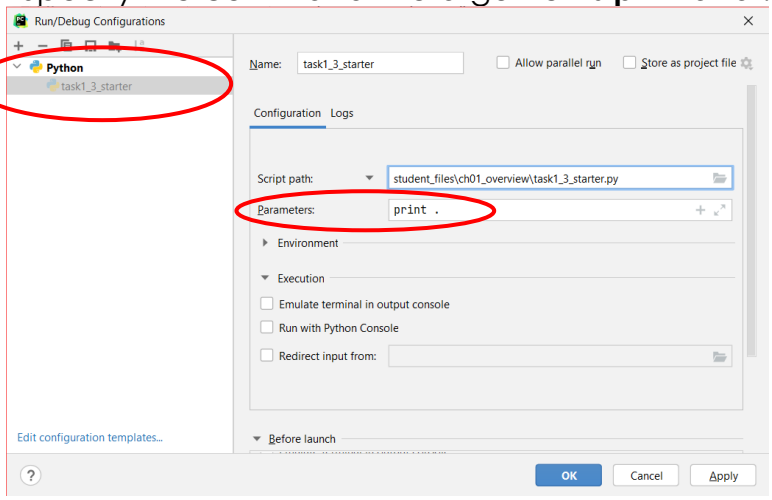
```
python task1_3_starter.py print .
```

Note: you may need to type python3.11, python3, or python3.x (etc.).

2. Using PyCharm: Run the program without arguments first. Then, select Run > Edit Configurations...

   Select the task1_3_starter entry on the left. On the right side, locate the input box called "Script parameters:"

   Specify the command line arguments **print** and **.**



   Click Ok and then run the script.

Task 1-4
Lists and Sorting Using glob()

Overview

This exercise displays files (not directories) from largest to smallest file size when given a specified input directory to read from. It uses the lists and sorting techniques discussed in class to accomplish the task.

Perform an Initial Run

Open the task1_4_starter.py file and run it viewing the output. Return to the code. Iterate over the **`dir_contents`** checking each item to see if it is a file.

```
import glob
import os
dir_contents = []

path = '.'
match = '*'
for pathitem in glob.glob('/'.join([path, match])):
    dir_contents.append(pathitem)

print(dir_contents)

files = []

# put your solution here
```

```
for item in dir_contents:
    if os.path.isfile(item):
```

Create a List of File Names and File Sizes

Append the filename (the basename only) and the file size into the provided **`files`** list. Each entry in the list should store two things: the file name and the file size. Do this by appending both items into the list as a tuple (ex: `(filename, filesize)`).

```
import glob
import os
dir_contents = []

path = '.'
match = '*'
for pathitem in glob.glob('/'.join([path, match])):
    dir_contents.append(pathitem)

print(dir_contents)

files = []

# put your solution here
for item in dir_contents:
    if os.path.isfile(item):
        files.append((os.path.basename(item),
                      os.path.getsize(item)))
```

Sort the List(by Size)

With the newly created **`files`** list (containing filenames and sizes), sort the list creating a function (we'll use a lambda) that will sort based on file size.

```
import glob
import os
dir_contents = []

path = '.'
match = '*'
for pathitem in glob.glob('/'.join([path, match])):
    dir_contents.append(pathitem)

print(dir_contents)

files = []

# put your solution here
for item in dir_contents:
    if os.path.isfile(item):
        files.append((os.path.basename(item),
                      os.path.getsize(item)))

files.sort(key=lambda fileinfo: fileinfo[1],
           reverse=True)
```

Print the Sorted List

Print the resulting list.

```
for name, size in files:
    print(f'{name:<20}{size}')
```

That's it! Test it out.

Task 1-5
Working with Dictionaries

Overview

This exercise will read the words within a text file (alice.txt) and then place the words into a dictionary. It will count the word occurrences and display the top 100 most frequent words that are five letters or more.

Create a Dictionary to Store Words

```
wordcount = {}
```

Read Lines from the Entire File

Even though we've touched on files briefly in previous exercises, we haven't officially introduced working with them. So, for now, we'll ignore exception handling. Iterate over the file reading line-by-line. Break each line into individual words using the split() method as shown:

```
for line in open('alice.txt', encoding='utf-8'):
    words = line.split()
```

Store Words in the Dictionary

For each of the words, add them as the keys to the dictionary. The dictionary will store the number of occurrences of words. If the word is already in the dictionary, increment its count:

```
for word in words:
    if word in wordcount:
        wordcount[word] += 1
    else:
        wordcount[word] = 1
```

Sort the Dictionary Items Based on Occurrences (Frequency Most to Least)

A dictionary cannot be sorted, but dictionary.items() can. **dictionary.items()** returns a list of tuples in the form of:

[(word1, count1), (word2, count2), ...]

You will want to sort each item based on the count value. The count value in each case is the second item in each tuple, so the following key function should work:

```
key=lambda a:a[1]
```

"a" in this case is a tuple as shown in the discussion above.

Don't forget to sort in reverse order (descending):

```
sortedwords = sorted(wordcount.items(), key=lambda
a:a[1], reverse=True)
```

Obtain Only Words 5 Letters or Greater

sortedwords, from the previous step is a list of (word, count) tuples sorted in order of most-to-least frequent. Create a new list that only contains words that are 5 letters or greater. A list comprehension can do this for you. Can you create this on your own first? If you need help, look down at the bottom of the page.

Finally, the list is sorted from most frequent to least. Use slicing to print the top 100 items in the list. Again, can you do this on your own?

```
five_letters = [(word, count) for word, count in
                sortedwords if len(word) >= 5]

print(five_letters[:100])
```

Task 2-1
Files and Exception Handling

Overview

This exercise requires reading from multiple data files. It also incorporates error handling and the use of context managers. The exercise retrieves baseball player salaries for a year specified by user input. To accomplish this task, two files will be read: Salaries.csv and People.csv. The first contains salary information that must be sorted, the other file contains the names and IDs of players.

Prompt User for a Year to Retrieve

Ask the user for which year they would like to search for salaries. Valid years are 1985 - 2016.

```
def salary_sort(sal_record):
    return sal_record.salary

year_str = input('Enter a year (1985-2016): ')
```

Read Salary Data into a NamedTuple

Open the Salaries.csv file by joining the working_dir and salaries_filename using os.path.join(). Using a with control, open the file as shown. We'll add some exception handling to deal with any errors (though we are just printing them). Add the boldfaced code shown below.

```
def salary_sort(sal_record):
    return sal_record.salary


year_str = input('Enter a year (1985-2016): ')

filepath = os.path.join(working_dir, salaries_filename)
try:
    with open(filepath, encoding='utf-8') as f_sal:
        # more here in a moment

except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()
```

Next, within the '*with*' statement, read the first line from the salaries.csv file, which is a header. We'll use this line to create a namedtuple object to how the records.

```
filepath = os.path.join(working_dir, salaries_filename)
try:
    with open(filepath, encoding='utf-8') as f_sal:
        header = f_sal.readline().strip().split(',')
        SalaryRecord = namedtuple('SalaryRecord', header)

        # more here in a moment

except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()
```

Now, continuing within the ***with*** control, read line-by-line from the file. Use *`split()`* to break apart the fields separated by a ',' (comma). Check that the year for that record is the same as the year the user is looking for. If so, store each record into a namedtuple and then into a list called *`salaries`*. This list is declared for you in the task2_1_starter.py file already.

Try to do this on your own before looking down at the bottom of this page.

Your solution should look something like this:

```
filepath = os.path.join(working_dir, salaries_filename)
try:
    with open(filepath, encoding='utf-8') as f_sal:
        header = f_sal.readline().strip().split(',')
        SalaryRecord =
                  namedtuple('SalaryRecord', header)

        for line in f_sal:
            data = line.strip().split(',')
            if year_str == data[0]:
                try:
                    data[4] = int(data[4])
                except ValueError:
                    data[4] = 0

                sal_rec = SalaryRecord(*data)
                salaries.append(sal_rec)

except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()
```

Read Player Data from the *People* File

Store each record in the provided ***players*** dictionary.

The *`players`* dictionary is already provided for you in the starter file. The keys for the dictionary should be the first field data[0] which represents the playerID.

Store the remainder of the record as the value within the dictionary. Here's a possible solution for this task:

```
filepath = os.path.join(working_dir, salaries_filename)
try:
    with open(filepath, encoding='utf-8') as f_sal:
        header = f_sal.readline().strip().split(',')
        SalaryRecord = namedtuple('SalaryRecord',
header)

        for line in f_sal:
            ...(not shown for brevity)...

except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()
```

```
people_filepath =
            os.path.join(working_dir, people_filename)
try:
    with open(people_filepath,
                        encoding='utf-8') as f_people:
        for line in f_people:
            data = line.strip().split(',')
            player_id, first_name, last_name =
                            data[0], data[13], data[14]
            players[player_id] =
                            (first_name, last_name)
except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()
```

Sort the Salaries

You should now have two data structures. One, a list, that contains namedtuples of playerIDs and salaries. The other, a dictionary keyed by playerIDs and holding player names.

Next, you will sort the salaries from largest to smallest. The starter file declared a sort() function for you already. It assumes a namedtuple is stored in the list and the salary field is called salary. Sort the salaries data structure (in-place) in descending order based on salary:

```
people_filepath = os.path.join(working_dir,
people_filename)
try:
    with open(people_filepath,
                          encoding='utf-8') as f_people:
        for line in f_people:
            data = line.strip().split(',')
            player_id, first_name, last_name =
                           data[0], data[13], data[14]
            players[player_id] =
                                 (first_name, last_name)
except IOError as err:
    print(err, file=sys.stderr)
    sys.exit()

salaries.sort(key=salary_sort, reverse=True)
```

Get the Top Salaries, Get the Player's IDs

Print a header to display the column names (Name, Salary, and Year).

Iterate over the records within the salaries data structure. Retrieve the playerID from each namedtuple.

Can you do this step on your own?
Look onto the next page to view one way to do this.

```
salaries.sort(key=salary_sort, reverse=True)

how_many = 10
print_header = ['Name', 'Salary', 'Year']
print('{0:35}{1:<20}{2:10}'.format(*print_header))

for salary_record in salaries[:how_many]:
    player_id = salary_record.playerID
```

Next, using the player ID obtained above, plug it into the dictionary to retrieve the player's first and last name. The 14th and 15th fields in the players record will contain the first and last names.

Once again, can you do this on your own before looking at the bottom for the solution?

```
how_many = 10
print_header = ['Name', 'Salary', 'Year']
print('{0:35}{1:<20}{2:10}'.format(*print_header))

for salary_record in salaries[:how_many]:
    player_id = salary_record.playerID
    (first_name, last_name) = players[player_id]
    name = first_name + ' ' + last_name
    salary = f'${salary_record.salary:<19,.2f}'
    year = salary_record.yearID

    print(f'{name:35}{salary:<}{year:<10}')
```

That's it! Test your app and check your results!

Task 3-1
Baseballs, Functions, and Modules

Overview

This exercise revisits the baseball salary exercise (Task 2-1). This time we'll create a function, called **load_data()** and place it into a separate module, called **baseball.py**. You should work from task3_1_starter.py and baseball.py. Both can be found within the ch03_functions directory.

Create the load_data() Function

Begin by either working from the task3_1_starter.py file or by working from your task2_1_starter.py file that you completed previously. If you use your own code, you can either copy it into task3_1_starter.py or copy the entire task2_1_starter.py file into ch03_functions. The option is up to you.

Create a load_data() function in baseball.py. While the method signature is up to you, the following is suggested:

```
def load_data(salaries_filepath,
              people_filepath, input_year='1985'):
  # migrate previous code (with changes needed) here
```

A possible version of baseball.py is shown on the next page:

```
from collections import namedtuple
import sys

def salary_sort(sal_record):
    return sal_record.salary

def load_data(salaries_filepath, people_filepath,
                                                input_year='1985'):
    salaries = []
    players = {}
    top_sals = []

    try:
        with open(salaries_filepath, encoding='utf-8') as f_sal:
            header = f_sal.readline().strip().split(',')
            SalaryRecord = namedtuple('SalaryRecord', header)

            for line in f_sal:
                data = line.strip().split(',')
                if input_year == data[0]:
                    try:
                        data[4] = int(data[4])
                    except ValueError:
                        data[4] = 0

                    sal_rec = SalaryRecord(*data)
                    salaries.append(sal_rec)
    except IOError as err:
        print(err)
        sys.exit()

    try:
        with open(people_filepath, encoding='utf-8') as
f_people:
            for line in f_people:
                data = line.strip().split(',')
                player_id, first_name, last_name = data[0],
                                                   data[13], data[14]
                players[player_id] = (first_name, last_name)
    except IOError as err:
        print(err)
        sys.exit()
```

```
    salaries.sort(key=salary_sort, reverse=True)

    for sal_info in salaries:
        year = sal_info.yearID
        salary = sal_info.salary
        playerid = sal_info.playerID
        player_data = players.get(playerid)
        if player_data:
            first_name, last_name = player_data
            top_sals.append((first_name, last_name, salary,
year))

    return top_sals
```

Set the PYTHONPATH

If you run the solution within PyCharm, you will not need to set a PYTHONPATH. If you run this from a command/terminal window, you will need to place the student_files on the PYTHONPATH environment variable as described in our course manual.

As a reminder, within Windows, open your Control Panel, locate the System icon, then select Advanced System Settings. From there, choose the 'Advanced' tab and then locate the Environment Variables button near the bottom. In the user variables section, add a variable called PYTHONPATH with a value to points to the location of your student_files folder.

Import the baseball Module and Invoke the Function

Your baseball.py module will need to be imported into the task3_1_starter.py file to be used. The code needed is shown on our slide in the manual, but here it is again:

```
import baseball
```

or

```
from ch03_functions import baseball
```

Modify your main file (task3_1_starter.py) to invoke the newly imported function.

```
import os

import baseball

working_dir = '../../resources/baseball/'
people_filename = 'People.csv'
salaries_filename = 'Salaries.csv'

year_str = input('Enter a year (1985-2016): ')

sal_filepath =
           os.path.join(working_dir, salaries_filename)
people_filepath =
           os.path.join(working_dir, people_filename)

top_sals = baseball.load_data(sal_filepath,
people_filepath, year_str)


how_many = 10
print_header = ['Name', 'Salary', 'Year']
print('{0:35}{1:<20}{2:10}'.format(*print_header))
```

```
for salary_info in top_sals[:how_many]:
    first_name = salary_info[0]
    last_name = salary_info[1]
    name = first_name + ' ' + last_name
    salary = f'${salary_info[2]:<19,.2f}'
    year = salary_info[3]
    print(f'{name:35}{salary:<}{year:<10}')
```

Test Your Solution

That's it! Test it out!

Task 4-1
Word Counters, Classes, and Properties

Overview

In this exercise, you will modify the provided WordCounter class used to count the words in a file by creating two properties. You will then test out your properties to make sure they work as expected.

Create the *result* Property

Work from **`task4_1_starter.py`** in the *ch04_oo*. Begin by making the result() function a property. This step is very simple as it only requires adding the @property decorator above the function and then removing the parentheses in the code where results() is called. Do this as shown below:

```
class WordCounter:
    def __init__(self, filepath, min_wordsize=1,
                 max_results=10, encoding='utf-8'):
        self.word_dict = defaultdict(int)
        ...details not relevant...

    @property
    def results(self):
        with open(self.filepath, encoding=
                                  self.encoding) as f:
            ...details not relevant...


sample_file = '../../resources/gettysburg.txt'
counter = WordCounter(sample_file, min_wordsize=5)
print(counter.results, counter.word_dict)
```

Create the *min_wordsize* Property

Convert the min_wordsize into a property. This means you will need to create the "getter" and "setter" functions.

Can you do this on your own before looking below?

```
class WordCounter:
    def __init__(self, filepath, min_wordsize=1,
                 max_results=10, encoding='utf-
8'):
        self.word_dict = defaultdict(int)
        ...details not relevant...

    @property
    def min_wordsize(self):
        return self._min_wordsize

    @min_wordsize.setter
    def min_wordsize(self, wordsize):
        self._min_wordsize = wordsize
        if self._min_wordsize <= 0:
            self._min_wordsize = 1


    @property
    def results(self):
        with open(self.filepath, encoding=
                                  self.encoding) as f:
            ...details not relevant...
```

Test it Out!

Test out your newly created properties to see if they behave properly. Try out a min_wordsize value of 0, for example.

```
sample_file = '../../resources/gettysburg.txt'
counter = WordCounter(sample_file, min_wordsize=0)
print(counter.results, counter.word_dict)
```

Task 5-1
Walking Directories and Copying Files

Overview

In this exercise, you will write a script that walks the entire student_files using **os.walk()**. You are searching for files that end with .jpg (Hint: there are two within the student files somewhere). When you find them, you should copy them into the student_files/resources/images folder.

Call os.walk()

Within **task5_1_starter.py**, invoke os.walk().

```
for curdir, subdirs, files in os.walk(root_directory):
```

Watch Out for the Destination Directory!

To avoid including our destination directory (the place where we are copying files to) in the walk, we will need to check that the current directory is not the 'resources/images' directory:

```
for curdir, subdirs, files in os.walk(root_directory):
    path = os.path.abspath(curdir)
    if path != dstpath:
```

Check for .jpg Files

If the current directory and destination directories are different, then we can check for .jpg files now:

```
for curdir, subdirs, files in os.walk(root_directory):
    path = os.path.abspath(curdir)
    if path != dstpath:
        for filename in files:
            if filename.lower().endswith('.jpg'):
```

Perform the Copy

You've found a .jpg file! Now copy it:

```
for curdir, subdirs, files in os.walk(root_directory):
    path = os.path.abspath(curdir)
    if path != dstpath:
        for filename in files:
            if filename.lower().endswith('.jpg'):
                print(f'Found: {filename} in {path}')
                filepath = os.path.join(path, filename)
                shutil.copy(filepath, dstpath)
```

That's it--Test it Out!

Test out your solution.

Task 5-2
Using pathlib.Path

Overview

This exercise will refactor the previous task (Task 5-1) to incorporate the **`Path`** object from the **`pathlib`** module. It should replace the need to use the **`os`** module. You should work from the provided starter file (task5_2_starter.py).

Create a Path() Object of the Root Directory

Wrap the student_files directory (we are calling the root_directory) with a pathlib.Path() object.

```
from pathlib import Path
import shutil

root_directory = Path('..')
dst = 'resources/images'
copied = 0
```

Create a Path() Object of the dst Directory

Repeat step 1 above, this time wrapping the dst directory ('resourse/images') in a Path() object. You can use the root_directory as a starting reference.

```
from pathlib import Path
import shutil

root_directory = Path('../..')
dst = root_directory / 'resources/images'
```

Iterate the Root Directory

Using the Path() object's **glob()** or **rglob()** method, iterate the **`root_directory`** and all sub-directories.

```
from pathlib import Path
import shutil

root_directory = Path('../..')
dst = root_directory / 'resources/images'
copied = 0

for pathitem in root_directory.glob('**/*.jpg'):
```

Handle Exceptions, Perform a File Copy

Within the newly created for-loop, create an exception handler (try-except block) to handle **`shutil.SameFileError`** messages. Within the try block, add a copy to perform a shutil.copy(). Optionally, display a message that the file was copied.

```
from pathlib import Path
import shutil

root_directory = Path('../..')
```

```
dst = root_directory / 'resources/images'
copied = 0

for pathitem in root_directory.glob('**/*.jpg'):
    try:
        shutil.copy(pathitem, dst)
        print(f'Match: {pathitem}. Copying...')

    except shutil.SameFileError as err:
        pass
```

Count the Number of Files Copied

Use the provided **copied** variable to count how many times a file is copied. Outside of the for-loop, display the **copied** variable.

```
from pathlib import Path
import shutil

root_directory = Path('../..')
dst = root_directory / 'resources/images'
copied = 0

for pathitem in root_directory.glob('**/*.jpg'):
    try:
        shutil.copy(pathitem, dst)
        print(f'Match: {pathitem}. Copying...')
        copied += 1
    except shutil.SameFileError as err:
        pass

print(f'Copied: {copied} files to dstpath')
```

This is our final version. Test it out!

Task 5-3
Parsing XML

Overview

This exercise will capture and display the contents of a live XML RSS feed. Results are stored in a list of namedtuples. Work from the **`task5_3_starter.py`** file within the **`ch05_std_lib`** folder. To shorten development time, the XML has already been retrieved and the document has been parsed for you.

Iterate the <item> Element. Obtain <title>, <description>, and <pubDate> Contents

The tree object should have already been created by parsing the XML for us. Iterate over the tree object by finding all the <item> elements within it. Use findall() as iter() is good only in the case of top-level elements.

```
items = []

for item in tree.findall('.//item'):
    title = item.find('.//title').text
    descriptionFull = item.find('.//description').text
    pubDateStr = item.find('.//pubDate').text
```

Place Items into a Namedtuple

The items obtained above can be placed into a namedtuple. The namedtuple has already been created for you:

```
items = []
for item in tree.findall('.//item'):
    title = item.find('.//title').text
    descriptionFull = item.find('.//description').text
    pubDateStr = item.find('.//pubDate').text
    items.append(
              Item(title, descriptionFull, pubDateStr))
```

Place the Namedtuple into a List

We'll place the Item() namedtuple into a list. Let's add the namedtuple (each iteration) into a list:

```
items = []
for item in tree.findall('.//item'):
    title = item.find('.//title').text
    descriptionFull = item.find('.//description').text
    pubDateStr = item.find('.//pubDate').text
    items.append(Item(title, descriptionFull,
pubDateStr))
```

Display the Results (Your Choice of Format)

Simply iterate over the items (list of Item namedtuples) and display the values from each one. The format is up to you. We'll use an f-string:

```
for item in items:
    print(f'{item.title} \n   {item.description}... \n
{item.pubDate}')
```

Optional Formatting (Truncate the Description and Convert the Date String)

The description contains lots of HTML and the dates have currently been left as strings. Let's truncate the description (up the beginning of the HTML) and convert the date strings into datetime objects:

```
items = []

for item in tree.findall('.//item'):
    title = item.find('.//title').text
    descriptionFull = item.find('.//description').text
    description = descriptionFull.split('<')[0]
    pubDateStr = item.find('.//pubDate').text
    pubDate = datetime.strptime(pubDateStr,
                                '%a, %d %b %Y %H:%M:%S %z')
    items.append(Item(title, description, pubDate))

for item in items:
    print(f'{item.title} \n
{item.description[:70]}... \n
{item.pubDate.strftime("%b %d, %Y")}')
```

This is our final version. Test it out!

Task 6-1
JSON, Requests, and HTML Parsing

Overview

This exercise creates two classes.

First, you will create an Event class to hold retrieved JSON event data from the **NASA Earth Observatory Natural Events Track API** (*eonet.gsfc.nasa.gov/api/v3/events*). Once data is retrieved, you will extract the all the "events" from the data storing them in the Event class. From these events, you will determine which types of *categories* are being tracked by NASA. In addition to this, you will list all the wildfires currently being tracked. Finally, you will retrieve a description of the first wildfire in the list of events by parsing a related HTML page (obtained within the JSON data).

Preparation:
Install BeautifulSoup and Requests

Perform the installation of the third-party tools, **beautifulsoup4** and **requests** (if needed). Generally, if you ran the requirements.txt file on day one (or allowed PyCharm to run it automatically), this step is already accomplished.

```
pip install beautifulsoup4
pip install requests
```

Create the Event Class

First, you will create a class, called *Event*, to hold a single event's data from the JSON response. It will have an __init__() with three parameters: *title*, *categories*, and *sources*. Create this class as shown by writing it below step 1 in the source file.

```
class Event:
    def __init__(self, title='',
                 categories=None, sources=None):
        self.title = title
        self.categories = [] if not categories else categories
        self.sources = [] if not sources else sources
```

Retrieve the JSON Data

Within the provided sync() method of the EONETData() class, use the requests module to retrieve the JSON data, convert it to a dict and save it into a variable called resp_dict.

```
resp_dict = requests.get(self.url).json()
```

Create the categories() Property

Given the categories property in the source code, complete it so that it returns all the possible category types. Properties should not do too much work or be expensive to call. These two properties are borderline. While they only return a value, they do have to create this value. Arguably, the property could be designed as a standard method instead.

Iterate over the *self.events* attribute, then iterate over the *event.categories* attribute. Add the category.get('title') into the provided set. Do this as shown:

.

```
@property
def categories(self):
    categories = set()

    for event in self.events:
        for category in event.categories:
            categories.add(category.get('title'))

    return categories
```

Create the fires() Property

Given the fires property, complete it so that it returns all the fire (Wildfires) event types. Iterate over *self.events*, then iterate over each event.categories (in each event). Check if the category.get('title') is the value 'Wildfires' and if so, add it to the fires list. Do this as shown:

```
@property
def fires(self):
    fires = []

    for event in self.events:
        for category in event.categories:
            if category.get('title') == 'Wildfires':
                fires.append(event)

    return fires
```

Instantiate an EONETData() Object

Within the provided try-except block, remove the pass statement and then instantiate an EONETData() object. Since the default value is provided, it doesn't need any arguments passed to it.

```
try:

    eonet = EONETData()

except requests.RequestException as err:
    print(f'{type(err).__name__}: {err}',
file=sys.stderr)
    sys.exit(42)
```

How Many Events Are There?

Using the EONETData() object from the previous step and the len() function, determine how many total events NASA is tracking.

```
print(f'NASA tracking {len(eonet.events)} total events.')
```

Get the Different Categories

Determine all the different category types. Do this by executing the EONETData() object's categories property (created in step 3).

```
print(f'\nSeveral categories found across all events:
{list(eonet.categories)}')
```

Display All of the WildFire Names (Titles)

Display a list of all the fires. Do this by executing the EONETData() object's fires property (created in step 4).

```
print('\nDisplaying fires tracked:')
fires = eonet.fires
print([fire.title for fire in fires])
```

Create a Volcanoes Property

Return to the EONet class and add a property that is nearly identical to the one created in step 4 for fires except this time it will be for volcanoes:

```
@property
def volcanoes(self):
    volcanoes = []
    for event in self.events:
        for category in event.categories:
            if category.get('title') == 'Volcanoes':
                volcanoes.append(event)
    return volcanoes
```

Get the Names of Volcanoes Tracked

Repeat step 8, this time for volcanoes:

```
print('\nDisplaying all volcanoes tracked:')
volcanoes = eonet.volcanoes
print([volcano.title for volcano in volcanoes])
```

Get a Description of the First Volcano

Using the URL associated with the first volcano being tracked, get a description of it using the provided selector:

```
selector = '#ProfileHolocene div.tabbed-box p.tab'
if volcanoes:
    volcano = volcanoes[0]
    url = volcano.sources[0].get('url')
    print(f'The first volcano tracked is {volcano.title}.')
    print(f'The URL for the volcano home page is {url}.')
    print('Retrieving the volcano home page...')
    volcano_html = requests.get(url).text
    print('Parsing the volcano home page...')
    volcano_soup = BeautifulSoup(volcano_html, 'html.parser')
    print('Selecting the volcano description...')
    results = volcano_soup.select(selector)
    print(results[0].text)
```

That's it! Test it out by running it within PyCharm.

Task 7-1
Regexes and Alice in Wonderland

Overview

Revisit the alice.txt exercise from earlier in the course (Task 1-5 previously). Remove capitalization and punctuation from words stored in the dictionary keys. This should result in a different answer for the most occurring word this time.

Search Keys for Punctuation

This task only requires one line of code added to the solution. It will scan keys added to the dictionary and remove unnecessary punctuation. Use re.sub() and the regex shown below. Perform the operation before items are entered into the dictionary.

```
r'[.!\?,:";\']'
```

Attempt this on your own before looking at the bottom of the page.

Answer:

```
word = re.sub(r'[.!\?,:"]','', word.lower())
```