**Python Mega-Course: 10 Apps Notes:**

Notes taken for “The Python Mega Course: Build 10 Real World Applications” on Udemy, taught by Ardit Sulce.

**List of Apps:**

* **App 1**: Web Mapping with Python: Interactive Mapping of Population and Volcanoes
* **App 2**: Controlling the Webcam and Detecting Objects
* **App 3 (part 1)**: Data Analysis and Visualization with Pandas and Matplotlib
* **App 3 (part 2)**: Data Analysis and Visualization - In-Browser Interactive Plots
* **App 4**: Web Development with Flask - Build a Personal Website
* **App 5**: GUI Apps and SQL: Build a Book Inventory Desktop GUI Database App
* **App 6**: Mobile App Development: Build a Feel-Good App
* **App 7**: Web-Scraping - Scraping Properties for Sale from the Web
* **App 8**: Flask and PostGreSQL - Build a Data Collector Web App
* **App 9**: Django & Bootstrap Blog and Translator App
* **App 10**: Build a Geography Web App with Flask and Pandas

Notes taken by Travis Rillos.

**Table of Contents**

[Section 1 – Welcome: 5](#_Toc111385468)

[Course Introduction: 5](#_Toc111385469)

[Section 2 – Getting Started with Python: 5](#_Toc111385470)

[Section Introduction: 5](#_Toc111385471)

[Section 3 – The Basics: Data Types: 5](#_Toc111385472)

[Python Interactive Shell: 5](#_Toc111385473)

[Terminal: 5](#_Toc111385474)

[Data Type Attributes: 6](#_Toc111385475)

[How to Find Out What Code You Need: 6](#_Toc111385476)

[What Makes a Programmer a Programmer: 6](#_Toc111385477)

[How to Use Datatypes in the Real World: 6](#_Toc111385478)

[Section 4 – The Basics: Operations with Data Types: 7](#_Toc111385479)

[More Operations with Lists: 7](#_Toc111385480)

[Accessing List Items: 7](#_Toc111385481)

[Accessing List Slices: 7](#_Toc111385482)

[Accessing Items and Slices with Negative Numbers: 7](#_Toc111385483)

[Accessing Characters and Slices in Strings: 8](#_Toc111385484)

[Accessing Items in Dictionaries: 8](#_Toc111385485)

[Tip: Converting Between Datatypes: 9](#_Toc111385486)

[Section 5: The Basics: Functions and Conditionals: 10](#_Toc111385487)

[Creating Your Own Functions: 10](#_Toc111385488)

[Intro to Conditionals: 10](#_Toc111385489)

[If Conditional Example: 11](#_Toc111385490)

[Conditional Explained Line by Line: 11](#_Toc111385491)

[More on Conditionals: 11](#_Toc111385492)

[Elif Conditionals: 11](#_Toc111385493)

[Section 6: The Basics: Processing User Input: 12](#_Toc111385494)

[User Input: 12](#_Toc111385495)

[String Formatting: 12](#_Toc111385496)

[String Formatting with Multiple Variables: 13](#_Toc111385497)

[More String Formatting: 13](#_Toc111385498)

[Cheatsheet: Processing User Input: 14](#_Toc111385499)

[Section 7: The Basics: Loops: 15](#_Toc111385500)

[For Loops: How and Why: 15](#_Toc111385501)

[Dictionary Loop and String Formatting: 15](#_Toc111385502)

[While Loops: How and Why: 15](#_Toc111385503)

[While Loop Example with User Input: 16](#_Toc111385504)

[While Loops with Break and Continue: 16](#_Toc111385505)

[Cheatsheet: Loops: 16](#_Toc111385506)

[Section 8: Putting the Pieces Together: Building a Program: 17](#_Toc111385507)

[Section Introduction: 17](#_Toc111385508)

[Problem Statement: 17](#_Toc111385509)

[Approaching the Problem: 17](#_Toc111385510)

[Building the Maker Function: 18](#_Toc111385511)

[Constructing the Loop: 19](#_Toc111385512)

[Making the Output User-Friendly: 20](#_Toc111385513)

[Section 9: List Comprehensions: 21](#_Toc111385514)

[Section Introduction: 21](#_Toc111385515)

[Simple List Comprehension: 21](#_Toc111385516)

[List Comprehension with If Conditional: 22](#_Toc111385517)

[List Comprehension with If-Else Conditional: 23](#_Toc111385518)

[Cheatsheet: List Comprehensions: 24](#_Toc111385519)

[Section 10: More About Functions: 25](#_Toc111385520)

[Functions with Multiple Arguments: 25](#_Toc111385521)

[Default and Non-default Parameters and Keyword and Non-keyword Arguments: 25](#_Toc111385522)

[Functions with an Arbitrary Number of *Non-keyword* Arguments: 26](#_Toc111385523)

[Functions with an Arbitrary Number of *Keyword* Arguments: 27](#_Toc111385524)

[Section 11: File Processing: 28](#_Toc111385525)

[Section Introduction: 28](#_Toc111385526)

[Processing Files with Python: 28](#_Toc111385527)

[Reading Text from a File: 28](#_Toc111385528)

[File Cursor: 29](#_Toc111385529)

[Closing a File: 29](#_Toc111385530)

[Opening Files Using “with”: 29](#_Toc111385531)

[Different Filepaths: 30](#_Toc111385532)

[Writing Text to a File: 30](#_Toc111385533)

[Appending Text to an Existing File: 31](#_Toc111385534)

[Cheatsheet: File Processing: 32](#_Toc111385535)

[Section 12: Modules: 33](#_Toc111385536)

[Section Introduction: 33](#_Toc111385537)

# Section 1 – Welcome:

## Course Introduction:

* Just an overview.
* This course will include how to program with Python from scratch, so I may end up skipping a lot of notes for the first 10 sections or so.
* There are **39 Sections**.
* There’s a Discord channel: <https://discord.gg/QWArvbdZVZ>

# Section 2 – Getting Started with Python:

## Section Introduction:

* Sounds like we use VSCode for this class. Sweet.

# Section 3 – The Basics: Data Types:

## Python Interactive Shell:

* For Windows, run **py -3** in the terminal to start the interactive shell.
* Useful for testing some throwaway code; interactive shell doesn’t save code.
* Creating .py files is better for creating reusable code.

## Terminal:

* Tip about splitting the terminal in two. This way we can run both the **powershell terminal** and the **Python Interactive Shell** side-by-side.
* This allows us to run test code in the interactive code and run .py code in the terminal.

## Data Type Attributes:

* Showed a useful command, **dir()**, which can be used very effectively in the Interactive Shell to find out what operations can be performed on a given subject (methods or properties).
  + Running **dir(list)** shows everything that can be performed on a list.
  + Running **dir(int)** shows everything that can be performed on an integer.
* He used the example of running **dir(str)** to see what can be performed on a string, chose **“upper”** from the list, then ran **help(str.upper)** to find out what it does.
  + This showed that “upper” is a method, which “Returns a copy of the string converted to uppercase”.
* Note: Functions follow the naming convention **function()** while methods follow the naming convention **.method()**.

## How to Find Out What Code You Need:

* To find a complete list of built-in functions, run **dir(\_\_builtins\_\_)**. These are functions that aren’t attached to a specific data type.
* We didn’t find an “average” or “mean” function, but there was a “**sum**” function. Between that and **len**, we can calculate an average for a list of floats.

## What Makes a Programmer a Programmer:

* Three things you need to know to make any program:
  + Syntax
  + Data Structures
  + Algorithm

## How to Use Datatypes in the Real World:

* In our example of creating a Dictionary of student names and grades, would we manually create this dictionary in the real world? Unlikely. The data would be stored in something like an Excel file.
* There are ways to automatically input data from an Excel file into Python.
* We will be doing this later in the course.

# Section 4 – The Basics: Operations with Data Types:

## More Operations with Lists:

* Went over a few methods such as **.append()**, **.index()**, and **.clear()**. Pretty basic stuff.
* Used **dir(list)** and **help(list.append)** etc to see what all can be done to lists.

## Accessing List Items:

* In our “basics.py” with the list “monday\_temperatures” in it, we used **monday\_temperatures.\_\_getitem\_\_(1)** to get the item at Index 1, which was 8.8.
* He then showed that instead of that, we can just use **monday\_temperatures[1]** and we get the same result.
* The version with the double underscores (“\_\_getitem\_\_(1)”) is probably the private method within the function, that the “**[1]**” syntax calls to.

## Accessing List Slices:

* To access a portion of a list **monday\_temperatures = [9.1, 8.8, 7.5, 6.6, 9.9]**, we can use the syntax:
  + **monday\_temperatures[1:4]**
  + To find the items at index 1, index 2, and index 3.
* We can also use **monday\_temperatures[:2]** to get every item before index 2, or the first two items.
* A similar shortcut, **monday\_temperatures[3:]** gives us the values from index 3 to the end of the list.

## Accessing Items and Slices with Negative Numbers:

* Get last item of list with **monday\_temperatures[-1]**. Super basic, but super useful.
  + In this case, running **monday\_temperatures[-5]** gives us the first item again.
* Running **monday\_temperatures[-2:]** with a colon gives us everything from the second-to-last item to the end of the list, or the last two items of the list.

## Accessing Characters and Slices in Strings:

* Strings have the exact same indexing system as lists (duh).
* We can also index a string that’s part of a list:
  + **monday\_temperatures = [‘hello’, 1, 2, 3]**
  + **monday\_temperatures[0]**
    - 🡪 ‘hello’
  + **monday\_temperatures[0][2]**
    - 🡪 ‘l’

## Accessing Items in Dictionaries:

* Started with the dictionary **student\_grades = {“Mary”: 9.1, “Sim”: 8.8, “John”: 7.5}** and input that in the Python interactive shell.
* Running **student\_grades[1]** gives us **KeyError: 1** because the dictionary doesn’t have a key called 1.
* However, running **student\_grades[“Sim”]** gives us 8.8.
* Instead of integers, dictionaries have **keys** as their indexes.
* He gave an example of why this can be very useful by writing a short English-to-Portuguese translation dictionary, then running **eng\_port[“sun”]** to output “**sol**”.

## Tip: Converting Between Datatypes:

Sometimes you might need to convert between different data types in Python for one reason or another. That is very easy to do:

**From tuple to list:**

1. >>> cool\_tuple = (1, 2, 3)
2. >>> cool\_list = list(cool\_tuple)
3. >>> cool\_list
4. [1, 2, 3]

**From list to tuple:**

1. >>> cool\_list = [1, 2, 3]
2. >>> cool\_tuple = tuple(cool\_list)
3. >>> cool\_tuple
4. (1, 2, 3)

**From string to list:**

1. >>> cool\_string = "Hello"
2. >>> cool\_list = list(cool\_string)
3. >>> cool\_list
4. ['H', 'e', 'l', 'l', 'o']

**From list to string:**

1. >>> cool\_list = ['H', 'e', 'l', 'l', 'o']
2. >>> cool\_string = str.join("", cool\_list)
3. >>> cool\_string
4. 'Hello'

As can be seen above, converting a list into a string is more complex. Here str() is not sufficient. We need str.join(). Try running the code above again, but this time using str.join("---", cool\_list) in the second line. You will understand how str.join() works.

# Section 5: The Basics: Functions and Conditionals:

## Creating Your Own Functions:

* Started with an example from earlier in the course where we calculated our own average because there was no built-in function to do so:

student\_grades = [9.1, 8.8, 7.5]

mysum = sum(student\_grades)

length = len(student\_grades)

mean = mysum / length

print(mean)

* Rather than do this, we can wrap these calculations in our own mean function that can then be used on other lists as well.
* I added some exception handling (to only accept a list and only return a float) to the code he presented:

def mean(mylist: list) -> float:

    the\_mean = sum(mylist) / len(mylist)

    return the\_mean

student\_grades = [8.8, 9.1, 7.5]

print(mean(student\_grades))

* He also ran **print(type(mean), type(sum))** in the same code and showed that ***mean*** *was class ‘function’ and* ***sum*** *was class ‘builtin\_function\_or\_method’.*

## Intro to Conditionals:

* What if in the previous code, we passed a dictionary instead of a list?
  + In my case, my code has some error handling.
* We’d get an error because ‘+’ can’t be used on an ‘int’ and a ‘str’. Our function isn’t designed to process dictionaries, just lists. However, we can fix this with conditionals.

## If Conditional Example:

* Note: I’ll have to take my exception handling out for forcing the input to be a list. Don’t know how to accept two different input data types yet.

def mean(myinput) -> float:

    if type(myinput) == list:

        the\_mean = sum(myinput) / len(myinput)

    elif type(myinput) == dict:

        the\_mean = sum(myinput.values()) / len(myinput)

    else:

        print("Invalid input type. Must be list or dictionary")

    return the\_mean

monday\_temperatures = [8.8, 9.1, 9.9]

student\_grades = {"Mary": 9.1, "Sim": 8.8, "John": 7.5}

print(mean(student\_grades))

print(mean(monday\_temperatures))

* Added some basic exception handling in the **else** statement that he didn’t have. He just routed any list inputs straight into “else”.

## Conditional Explained Line by Line:

* In this video he just went through and explained what was going on line-by-line. Basic stuff.

## More on Conditionals:

* Stuff on Booleans, True/False, and how this works in conditionals.
* He mentioned the use of **if isinstance(myinput, dict)** as a useful bit of syntax. I should use that more often in my own code.
* He mentions that there are some very advanced reasons why the isinstance syntax is better to use, but that we won’t get into that until later in the course.

## Elif Conditionals:

* And yet I already used one in my earlier code. The structure of his course still makes sense for absolute beginners, but these first few sections are a bit of a slog.

# Section 6: The Basics: Processing User Input:

## User Input:

* We’re going to be taking user input in the form of a temperature, to run through a function.

def weather\_condition(temperature: float) -> str:

    if temperature > 7:

        return "Warm"

    elif temperature <= 7:

        return "Cold"

    else:

        return "Invalid input. Please enter a number."

**user\_input = float(input("Enter temperature: ")) 🡨 🡨 🡨**

print(weather\_condition(user\_input))

* Added some exception handling again.
* We had to make sure the input was converted to a float (or an int), or else the program will take the input in as a string by default.

## String Formatting:

* Now here’s some wildcard syntax I don’t see too often yet:

user\_input = input("Enter your name: ")

**message = "Hello %s!" % user\_input 🡨 🡨 🡨**

print(message)

* The <**%s**> and <**% user\_input**> in particular is an interesting way to go about inputting that name. An **f-string** would probably also work if I can remember the proper syntax for one.
* Oh wait, he did one:

user\_input = input("Enter your name: ")

message = "Hello %s!" % user\_input

**message = f"Hello {user\_input}" 🡨 🡨 🡨**

print(message)

* He noted that the f-string method works for Python 3.6 and above. The other method works for Python 2 and Python 3.
* You may want to program for an older version of Python, depending on the webserver you’re running it on.

## String Formatting with Multiple Variables:

* To use multiple variables, you more-or-less just add them on.

name = input("Enter your name: ")

surname = input("Enter your surname: ")

message = "Hello %s %s!" % (name, surname) **🡨 🡨 🡨**

message = f"Hello {name} {surname}!" **🡨 🡨 🡨**

print(message)

## More String Formatting:

There is also another way to format strings using the "{}".format(variable) form. Here is an example:

1. name = "John"
2. surname = "Smith"
4. message = **"Your name is {}. Your surname is {}".format(name, surname)**
5. print(message)

Output: Your name is John. Your surname is Smith

## Cheatsheet: Processing User Input:

In this section, you learned that:

* A Python program can get **user input** via the input function:
* The **input** **function** halts the execution of the program and gets text input from the user**:**

1. name = input("Enter your name: ")

* The input function converts any **input to a string**, but you can convert it back to int or float:

1. experience\_months = input("Enter your experience in months: ")
2. experience\_years = int(experience\_months) / 12

* You can also **format strings** with:

1. name = "Sim"
2. experience\_years = 1.5
3. print("Hi {}, you have {} years of experience".format(name, experience\_years))

Output: Hi Sim, you have 1.5 years of experience.

# Section 7: The Basics: Loops:

## For Loops: How and Why:

* For loop iteration. Basic.

## Dictionary Loop and String Formatting:

Here is an example that combines a dictionary loop with string formatting. The loop iterates over the dictionary and it generates and prints out a string in each iteration:

1. phone\_numbers = {"John": "+37682929928", "Marry": "+423998200919"}
3. for pair in phone\_numbers.items():
4. print(f"{pair[0]} has as phone number {pair[1]}")

And here is a better way to achieve the same results by iterating over keys and values:

1. phone\_numbers = {"John": "+37682929928", "Marry": "+423998200919"}
3. for key, value in phone\_numbers.items():
4. print(f"{key} has as phone number {value}")

In both cases, the output is:

John has as phone number +37682929928

Marry has as phone number +423998200919

## While Loops: How and Why:

* He showed an infinite loop for starters. Interesting choice.

## While Loop Example with User Input:

* Just a basic example to check if a username is correct.

username = ''

while username != 'pypy':

    username = input("Enter username: ")

## While Loops with Break and Continue:

* Same functionality as previous, but different method:

while True:

    username = input("Enter username: ")

    if username == 'pypy':

        break

    else:

        continue

* He says he prefers this method over the previous one because it gives you more control over the workflow. He also finds it more readable.

## Cheatsheet: Loops:

* We also have **while-loops**.The code under a while-loop will run as long as the while-loop condition is true:
  1. while datetime.datetime.now() < datetime.datetime(2090, 8, 20, 19, 30, 20):
  2. print("It's not yet 19:30:20 of 2090.8.20")

The loop above will print out the string inside print() over and over again until the 20th of August, 2090.

# Section 8: Putting the Pieces Together: Building a Program:

## Section Introduction:

* The purpose of this section is to fill in gaps in Python knowledge, to make everything work together.

## Problem Statement:

* He showed off just the output of a program called **textpro.py**.
* The program takes some basic input sentences and then reformats them with proper capitalization and punctuation.
* Input prompts end when the input is “**\end**”.

## Approaching the Problem:

* We’re going to look closely at the output (“It’s good weather today. How is the weather there? There are some clouds here.”).
* It’s good to have a very clear idea of what the output should be.
* We look at the output and figure out how it can be broken down into smaller tasks.
* We’re going to accomplish this with multiple functions.

## Building the Maker Function:

* We tested several methods in our Python interactive shell as we went along, to test that their functionality would work for us.
  + **“how are you”.capitalize()** gave us “How are you”
    - We wouldn’t use .title() here because that would capitalize (almost) every word.
  + **“how are you”.startswith((“who”, “what”, “where”, “when”, “why”, “how”))** checks the phrase against a tuple containing all our interrogative words. This is how we can decide whether a sentence should end with a **“?”** or not.
* Here’s what we had by the end of the lecture:

def sentence\_maker(phrase):

    interrogatives = ("who", "what", "where", "when", "why", "how")

    capitalized = phrase.capitalize()

    if phrase.startswith(interrogatives):

        return "{}?".format(capitalized)

    else:

        return "{}.".format(capitalized)

print(sentence\_maker("how are you"))

* We tested with the phrase “how are you” to check functionality, and it came back properly formatted:
  + 🡪 How are you?

## Constructing the Loop:

* We want to add the **user input** now, and we use a **while loop** to divide the flow of the program:

def sentence\_maker(phrase):

    interrogatives = ("who", "what", "where", "when", "why", "how")

    capitalized = phrase.capitalize()

    if phrase.startswith(interrogatives):

        return "{}?".format(capitalized)

    else:

        return "{}.".format(capitalized)

results = []

while True:

    user\_input = input("Say something: ")

    if user\_input == "\end":

        break

    else:

        results.append(sentence\_maker(user\_input))

print(results)

* Our outputs at this stage are still in the form of lists. Lists of phrases that have been properly formatted, but still lists. We want strings.
  + 🡪 [‘Weather is good.’, ‘How are you?’]

## Making the Output User-Friendly:

* Now we want to concatenate all these strings using the **.join()** method.
* The example he ran in the Python interactive shell was:
  + >>> **“-”.join([“how are you”, “good good”, “clear clear])**
  + 🡪 ‘how are you-good good-clear clear’
* The **.join()** method joins items together in a string, with whatever is in between the quotation marks separating the items:

def sentence\_maker(phrase):

    interrogatives = ("who", "what", "where", "when", "why", "how")

    capitalized = phrase.capitalize()

    if phrase.startswith(interrogatives):

        return "{}?".format(capitalized)

    else:

        return "{}.".format(capitalized)

results = []

while True:

    user\_input = input("Say something: ")

    if user\_input == "\end":

        break

    else:

        results.append(sentence\_maker(user\_input))

**print(" ".join(results))** 🡨 🡨 🡨

* Here we used **“ ”.join(results)** to turn the list of formatted phrases into a string, with a space in between them all.

# Section 9: List Comprehensions:

## Section Introduction:

* Primary difference between List Comprehensions and for-loops is that List Comprehensions are written in a single line while for-loops are written in multiple lines.
* They’re a special case of for-loops that are used when you want to construct a list.

## Simple List Comprehension:

* The first example here involves presenting a list of temperatures in Celsius, but without the decimal points. This is often down to save disk space.
* Here’s how a list of temperatures would be re-calculated to add decimal points using a for-loop:

temps = [221, 234, 340, 230]

new\_temps = []

for temp in temps:

    new\_temps.append(temp / 10)

print(new\_temps)

* However, there’s a neater way to accomplish this using just a single line of Python code:

temps = [221, 234, 340, 230]

new\_temps = [temp / 10 for temp in temps]

print(new\_temps)

* Much neater. There’s an in-line for-loop in the new\_temps list.

## List Comprehension with If Conditional:

* Similar to previous, but in this case we include some invalid data (-9999). We want to ignore this one.

temps = [221, 234, 340, -9999, 230]

new\_temps = [temp / 10 for temp in temps if temp != -9999]

print(new\_temps)

**More Examples:**

* Define a function that takes a list of both strings and integers and only returns the integers.
  + Ex.: **foo([99, ‘no data’, 95, 94, ‘no data’])** returns **[99, 95, 94]**:

def foo(data):

new\_data = [item for item in data if isinstance(item, int)]

return new\_data

* Define a function that takes a list of numbers and returns the list containing only the numbers greater than 0.
  + Ex.: **foo([-5, 3, -1, 101])** returns **[3, 101]**:

def foo(data):

new\_data = [item for item in data if item > 0]

return new\_data

## List Comprehension with If-Else Conditional:

* If you want to add an **else** statement in list comprehension (such as “if number != -9999 else 0”) the order is a little different from what we’re used to in if-else conditionals.

temps = [221, 234, 340, -9999, 230]

new\_temps = [temp / 10 **if temp != -9999 else 0** **for temp in temps**] 🡨 🡨 🡨

print(new\_temps)

* Need to get used to this order more often.

**More Examples:**

* Define a function that takes a list of both numbers and strings, and returns numbers or 0 for strings:

def foo(data):

new\_data = [item if isinstance(item, int) else 0 for item in data]

return new\_data

* Define a function that takes a list containing decimal numbers as strings, then sums those numbers and returns a float:

def foo(data):

new\_data = [float(item) for item in data]

return(sum(new\_data))

## Cheatsheet: List Comprehensions:

In this section, you learned that:

* A list comprehension is an expression that creates a list by iterating over another container.
* A **basic**list comprehension:
  1. [i\*2 for i in [1, 5, 10]]

Output: [2, 10, 20]

* List comprehension with **if** condition:
  1. [i\*2 for i in [1, -2, 10] if i>0]

Output: [2, 20]

* List comprehension with an **if** **and** **else** condition:
  1. [i\*2 if i>0 else 0 for i in [1, -2, 10]]

Output: [2, 0, 20]

# Section 10: More About Functions:

## Functions with Multiple Arguments:

* Separate the parameters with a comma while defining the function (basic stuff).
* Calling the function will now take two arguments.

## Default and Non-default Parameters and Keyword and Non-keyword Arguments:

* Example of a function with “default parameters” set:
  + **def area(a, b = 6)**
  + You can also manually assign a new value for **b** even if there’s a default setting
* Example of function being called with “keyword arguments”:
  + **print(area(a = 4, b = 5)**
  + Also called “non-positional arguments”.
  + A “positional argument” would be where there’s no keyword and the position of the argument defines its meaning, i.e. **print(area(4, 5))**.
    - **print(area(b= 5, a = 4)** also works.

## Functions with an Arbitrary Number of *Non-keyword* Arguments:

* Some built-in functions take a specific number of arguments:
  + **len()** takes exactly 1 argument.
  + **isinstance()** takes exactly 2 arguments.
* Other built-in functions can take an arbitrary number of arguments:
  + **print()** can take any number of arguments.
* In this lecture, we’re going to create a function that can take any number of arguments when called.
* Do define a function like this, we use the syntax:
  + **def mean(\*args):**
  + “args” is a pretty standard name for this, that almost all Python programmers use.
  + If we simply **return** **args**, we get a tuple back that’s full of the arguments we passed in.
  + Note that keyword arguments would not work in this situation.

def mean(\*args): 🡨 🡨 🡨

    return sum(args) / len(args)

print(mean(1, 3, 4))

**More Examples:**

* Define a function that takes an indefinite number of strings and returns an alphabetically sorted list containing all the strings converted to uppercase:

def foo(\*args):

    words = [word.upper() for word in args]

    return sorted(words)

* Or:

def foo(\*args):

    words = []

    for word in args:

        words.append(word)

    return sorted(words)

## Functions with an Arbitrary Number of *Keyword* Arguments:

* In the previous case we defined our function with **def mean(\*args)**.
* The case with keyword arguments is similar:
  + **def mean(\*\*kwargs):** with “kwargs” being a standard convention.
  + However, this takes keyword arguments only. Unnamed arguments will cause an error.
  + Returning these arguments gives us a **dictionary** with the **keyword names** being the ‘**keys**’ and the **arguments** being the ‘**values**’.
  + Running **print(func(\*\*kwargs(a=1, b=2, c=3))** yields **{‘a’: 1, ‘b’: 2, ‘c’: 3}**.
* Functions with an arbitrary number of keyword arguments are *more rarely* used than functions with an arbitrary number of non-keyword arguments.

# Section 11: File Processing:

## Section Introduction:

* Storing data *outside* Python in external files.
* Text files, .csv files, databases.

## Processing Files with Python:

* He had created a text file called **fruits.txt** containing:
  + pear
  + apple
  + orange
  + mandarin
  + watermelon
  + pomegranate
* In the next lecture, we’ll use Python to *read* this file.

## Reading Text from a File:

* My Python file, **file-process.py** is in the same directory as my copy of **fruits.txt**.

myfile = open("fruits.txt")

print(myfile.read())

* The code to open this file is:
* The argument in the **open()** method is the filepath for the .txt file. In this case, just giving the name of the .txt file should be enough because both files are in the same directory.
* Note: I couldn’t get it to work at first, even though both files were in the same directory for Section 11. I ended up running “**pwd**” in bash and it turns out my ***working directory*** was one level up, so I ran “**cd**” to get into the directory both were saved in.

## File Cursor:

* The cursor starts at the first character of the file we’re reading in, and goes through to the end of the file.
* At the end of reading a file, the cursor is at the end of the file. Running **print(myfile.read())** on two or more lines of code won’t do anything.
* What you could do instead is to save **myfile.read()** into a variable, and then you can print out that variable multiple times instead.

myfile = open("fruits.txt")

content = myfile.read()

print(content)

print(content)

print(content)

## Closing a File:

* When you create a file object, a file object is created in RAM. It’s going to remain there until your program ends.

myfile = open("fruits.txt")

content = myfile.read()

myfile.close()

print(content)

* Therefore, it would be a good idea to close the file at the end of the program.
* However, there’s also a better way to do this, which we’ll cover in the next lecture.

## Opening Files Using “with”:

* Using the **with** method does all the opening, reading, and closing as a block:

with open("fruits.txt") as myfile:

    content = myfile.read()

print(content)

## Different Filepaths:

* For this, we’ll be moving **fruits.txt** to another directory.
* We need to add the filepath into our **open()** function:

with open("files/fruits.txt") as myfile:

    content = myfile.read()

print(content)

## Writing Text to a File:

* We started by running the **help(open)** function to see its attributes.
* The first two are most important: **file** and **mode=’r’** (meaning the default mode is “read”).

with open("files/vegetables.txt", "w") as myfile:

    myfile.write("Tomato\nCucumber\nOnion\n")

    myfile.write("Garlic")

* We’re going to create a new file, **vegetables.txt** using the “w” write option.
* Note: If the filename already exists, Python will overwrite the existing file.
* The special character **\n** is useful to make sure items are written on new lines.

**More Examples:**

* Define a function that takes a single string **character** and a **filepath** as parameters and returns the **number of occurrences** of that character in the file:

def foo(character, filepath):

count = 0

with open(filepath) as myfile:

content = myfile.read()

for char in content:

if char == character:

count += 1

else:

pass

return count

## Appending Text to an Existing File:

* We want to add two more lines to our existing **vegetables.txt** file. It currently has:
  + Tomato
  + Cucumber
  + Onion
  + Garlic
* If you look at the **help(open)** documentation and scroll down, you’ll see an option to set the mode argument to **“x”** (“create a new file and open it for writing”). Unlike the “w” option, this will not overwrite a file if it already exists.
* There’s also a mode argument **“a”** (“open for writing, appending to the end of the file if it exists”). We’re going to use this to add **Okra** to the list:

with open("files/vegetables.txt", "a") as myfile:

    myfile.write("Okra")

* Running this adds Okra to the end of the existing file, but not on a new line. The last line will read as “GarlicOkra”. There wasn’t a break-line (“**\n**”) in the existing file.
* To fix this, we change the code to:

with open("files/vegetables.txt", "a") as myfile:

    myfile.write("\nOkra")

* He then showed us an example of trying to append and *read* right after. However, since we set the mode to **“a”**, we can’t read, and we get an error.
* To get around this we look in the **help(open)** documentation and see an add-on option **“+”** (“open a disk file for updating (reading and writing)”).

with open("files/vegetables.txt", "**a+**") as myfile: 🡨 🡨 🡨

    myfile.write("\nOkra")

    content = myfile.read()

print(content)

* However, just running this doesn’t print anything out. We need to add something else as well: the **.seek(0)** method to put the cursor at the zero position again:

with open("files/vegetables.txt", "a+") as myfile:

    myfile.write("\nOkra")

    myfile.seek(0) 🡨 🡨 🡨

    content = myfile.read()

print(content)

* The cursor goes back to the beginning, and then reads down to the end of the file.

## Cheatsheet: File Processing:

In this section, you learned that:

* You can **read** an existing file with Python:

1. with open("file.txt") as file:
2. content = file.read()

* You can **create** a new file with Python and **write** some text on it:

1. with open("file.txt", "w") as file:
2. content = file.write("Sample text")

* You can **append** text to an existing file without overwriting it:

1. with open("file.txt", "a") as file:
2. content = file.write("More sample text")

* You can both **append and read** a file with:

1. with open("file.txt", "a+") as file:
2. content = file.write("Even more sample text")
3. file.seek(0)
4. content = file.read()

# Section 12: Modules:

## Section Introduction:

* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A
* A