# Intro

What the documentation is about

Lists of operators structured as operator—tab—meaning of operator

Need to look at for future revisions

Heading link

Glossary term

Python

Python instructions on what goes here

Python code blocks

## Courses to Take Notes from

* <https://docs.python.org/3.8/>
* <https://www.python.org/dev/peps/pep-0008>
* <https://www.python.org/dev/peps/pep-0257>
* Python Projects
* Python: XML, JSON, and the Web
* Pandas Essential Training
* Data Ingestion with Python
* Advanced Python
* Learning the Python 3 Standard Library
* Python Standard Library Essential Training
* Using Python with Excel (maybe)
* Using Python for Automation (maybe)
* Python GUI with Tkinter
* <https://realpython.com/beginners-guide-python-turtle/>
* Database Clinic: MySQL

# The Basics: Structure of a Python Program

## Program Opening

#!path to Python interpreters

# -\*- coding: encoding -\*-

# Git repository

# Other introductory information

* Shebang (#!) line needs to start scripts to be run from Unix command line
* Python files are encoded in UTF-8 by default; to encode in another format, add the coding line above and replace encoding with the desired encoding format’s Python alias
  + The list of standard encodings and their aliases is [here](https://docs.python.org/3.8/library/codecs.html#standard-encodings)
  + If there’s no shebang line, this line needs to be the first one in the program
* Python comments
  + Single line and in-line comments created with hash #
  + Multi-line comments enclosed by three single or double quotes
* Put if being tracked by Git here
* Listing of beautiful comment keywords here?

## Import Statements

import module or package

import package.module

import package as alias

from package import module, class, function

* All modules and packages used by the program should be imported at the very beginning
  + Import statements are typically ordered by hh order here
  + If an imported module changes, for the change to be recognized by a running script, the module must be reloaded with module.reload()
* Modules can be from the Python Standard Library, the Python Package Index (PyPi), or another user-created Python file
  + When importing another file, if both files are in the same folder, module is the file name without the file extension
* To import part of a package or module, a dot operator can be used
  + Multiple items can be strung together via dot operators
  + All but the last item in the dot operator sequence must be packages; the last item can be a package or module
* Using as alias means all uses of package in the program use alias instead
  + Aliases are important if a module contains a method with the same name
* When using from package import something, package is not defined in the program
  + The variant from package import \* imports all names from package except ones beginning with an underscore; in this case, package is defined in the program
  + If the package’s init file has an \_\_all\_\_ attribute, the module names contained in that attribute determine what’s imported instead
* A dot operator sequence, aliases, and from package import something constructs can be used together in an import statement
* Adding “library” to glossary
* See Modules and Packages for more info
* Appendix B: Python Modules contains the modules and packages explicitly handled in this documentation

## Function Definitions

def function\_name (parameter1, parameter2 = default parameter2 value):

“”” docstring describing the function “””

body of function

return return value

* Functions are defined before the main program because they must be defined before being called
* Function code blocks are designated by indentation
* First line will always have parentheses even if the function has no parameters
* Functions always return a value, but the return keyword isn’t required—if not used, the function returns None
* Generators are functions that return multiple values
* Parameters
  + To supply a default value for a parameter, follow the parameter name with an assignment operator and the default value
    - If a parameter has a default value, a corresponding argument isn’t required when the function is called
    - Parameters without defaults are listed before parameters with defaults
  + Var-positional parameters are declared with \*args
    - Var-positional parameters allow for a variable length list of arguments
    - The collection provided in the corresponding argument is utilized as a tuple
  + Var-keyword parameters are declared with \*\*kwargs
    - Var-keyword parameters allow for a dictionary as an argument
      * The dictionary can serve as a set of keyword arguments
  + See Parameter Syntax for more info
* Functions can include the ability to raise standard Python errors—see Raising Errors in Functions and Classes
* The first line of the body of the function should be a docstring describing the function
  + String uses three double quotation marks, so it can span multiple lines
  + H
  + See Docstrings for more info
* h

## Class Definitions

class class\_name:

“”” docstring describing the class “””

# Basic Constructor Method

def\_\_init\_\_(self, parameter):

self.\_attribute = parameter

# Getter/Setter Methods

def attribute(self, NewValue = None):

if NewValue: return self.\_attribute = NewValue

return self.\_attribute

**# Other Methods**

def method1(self):

body of function

def method2(self, parameter corresponding to an argument):

body of function

* Classes are made up of methods, some of which are used to assign attributes
  + The first parameter, which represents the object, is traditionally named self
    - self doesn’t have a corresponding argument in the parentheses when the method is called—its argument is the object name the method is appended to via dot notation—see Methods and Dot Notation for more info
* Class code blocks are designated by indentation and start with a docstring describing the class
  + See Docstrings for more info
* Attributes are usually assigned to class objects via the \_\_init\_\_ method, which allows for assignment when each new object is created
  + The \_\_init\_\_ method is a special constructor (meaning the name has two underscores before and after it) for defining the attributes of the class
    - The parameters after self correspond to the arguments that will be needed when the object is initialized
    - Constructed attribute variables traditionally begin with an underscore to indicate they should be treated as private (not used outside the class code block)
    - Constructed attributes are accessed in the main program via getter methods, which are usually given the same name as the attribute sans underscore
      * Getter/Setter methods allow the getter method to also be used to change the value of the attribute—for more info, see Using a Constructor Method with Var-Keyword Parameters
* Class code blocks are designated by indentation
* Classes can include the ability to raise standard Python errors—see Raising Errors in Functions and Classes
* hh
* See Classes and Object-Oriented Programming for more information

### Create a Tkinter GUI

class GUI:

“””docstring”””

def \_\_init\_\_(self, root):

self.variable = ttk.widget(root, arguments)

self.variable.geometry(arguments)

ttk.widget(root, arguments).geometry(arguments)

def method(self):

some code involving changes to self.variable

# In the main program

root\_window = Tk()

GUI = GUI(root\_window)

root\_window.mainloop()

* Importing the tkinter module provides access to basic GUI creation functionality
  + The themed widgets in the ttk module within the tkinter module require importing the ttk module as well
* Creating the GUI as a class allows for it to be more readily incorporated into object-oriented programs—see Creating the GUI Class for more info
  + To use the class
    - The main GUI window is initialized with Tk()
    - An object of the GUI class is initialized with the main GUI window variable as the argument
    - The mainloop method is called on the main GUI window variable
* The GUI is comprised of widgets
  + Widgets exist in a hierarchy with the root window at the very top—all widgets but that root window must have a parent
    - Not all widgets will be visible—some will exist for organizational or layout purposes related to this hierarchical structure
    - Widgets that will not need to be referenced later can be created without being saved to a variable and not be garbage collected
      * These widgets will need to be geometry managed immdiately
  + See Widgets for more info and Tkinter Widgets for the methods for creating widgets
* Widgets won’t appear in the GUI until geometry management provides instructions for placement
  + The Pack Geometry Manager is the simplest, packing children along the edges of the parent and against one another
  + The Grid Geometry Manager divides the parent into a grid and uses the grid to place widgets
  + The Place Geometry Manager uses the parent as a coordinate plane and sets widgets at absolute or ratio-determined points in the plane
  + All of the children of a given widget should use the same geometry manager
* Event handling allows the GUI to react to user events
  + Each reaction to a user action appears in the GUI class as a method
  + The mainloop method which is called on the root window initiates the event loop, which is where the program waits until an action with an event handler is detected—at that point, it exits the loop, executes the handler function, and returns to the loop
    - If an event occurs while the program is running a handler function, the event is added to a queue to be drawn from when the program returns to the event loop
  + Commonly interactive widgets have a command attribute related to the execution of the common interaction
    - All widgets have a bind attribute that can be used for event handling
* See GUI Creation for more info

## Initializing Variables

* The assignment operator is an equals sign (=), and it makes the value before it equal to the value after it
  + Technically, assignment works with locations in memory and the contents of those memory locations
  + Multiple assignments can be done on a single line—the values on both sides are separated by commas, and the assignments are position-based
    - Syntax is a, b = 1, 0 which is the same as a = 1 and b = 0
  + Assignment within an expression is done with the walrus operator (:=)
* For this documentation, the variable names are not indicated as varying, but variables can actually be given any name that follows the following rules
  + H
* The underscore (\_) is the variable for the last printed expression when using the command line or IDLE

### Initialize Numeric Data Types

int\_variable = an integer

float\_variable = a decimal number

precise\_decimal\_variable = Decimal(‘a decimal number’)

* Numeric data types represent numbers
  + Integers (int) are whole numbers
  + Floating-point (float) are decimals that are precise but not accurate
  + hhh
* The decimal module allows for accurate calculations with its Decimal class
* List numeric data types, dates
* For more information on numeric and date data types, see Numeric Data Types

### Initialize Boolean and Null Values

true\_variable = True

false\_variable = False

null\_variable = None

* All three values above are keywords
* The bool types can be set directly but are also the result of an evaluated conditional expression
* None is the absence of a value
* See Boolean Values for more info on Booleans

### Initialize Strings

str\_variable = ‘string’

str\_variable = “string”

str\_variable = ‘’’

string on

multiple lines

‘’’

* Strings need to be surrounded by
  + Single quote
  + Double quote
  + Three single quotes (allows for multiple lines; newline character preserved)
  + Three double quotes (allows for multiple lines; newline character preserved)
* The type of quote used to denote a string can be used within the string by preceding it with a slash (\)
  + Preceding a character with a slash is called escaping a character

to\_format = “string {}”

* format() can use a string variable initialized with curly braces
  + The overall method call looks like to\_format.format(x) where x replaces the curly braces
  + Complete details on the format method are in String Formatting
* H
* See Strings for more info

### Initialize Collections and Data Structures

list\_variable = [list element 1, list element 2]

tuple\_variable = (tuple element 1, tuple element 2)

dictionary\_variable = {key 1: value 1, key 2: value 2}

dataframe\_variable =

* Lists and tuples are both sequences of other data types—the difference is that the former is mutable
* Items in a collection have the name structure variable[index]
  + For lists and tuples, index is the position number starting at 0
    - Numeric indexes are sliceable—Index Referencing has more details
  + For dictionaries, index is the key
* H
* See Collections and Dataframes for more info

### Initialize an Object of a Custom Type

class\_variable = class()

class\_variable = class\_with\_two\_attributes(argument1, argument2)

* An assignment operator is used to create an object of type class
* H
* See Classes and Object-Oriented Programming for more info

## Statements

* Expressions and Statements
  + Expressions return a value (a unit of evaluation)
  + Statements are effectively a line of code (a unit of execution)
  + Multiple expressions can be on a single line separated by semicolons—this is very rare
* Operator precedence is determined in the order below, then moving left to right
  + Mathematical operators in order of operations
    - Parentheses
    - Exponents
    - Unary positive and negative signs
    - Multiplication and division
    - Addition and subtraction
  + Bitwise operators—see Bitwise Operations for more info
    - Shifts
    - and
    - xor
    - or
  + Comparison, identity, and membership operators—see Boolean Operators for more info
  + Logical operators—see Boolean Operators for more info
    - not
    - and
    - or
* h

### User I/O

print(“Hello world!”)

user\_input = input(argument)

* User I/O is this documentation’s term for information output to the user or input by the user
* User I/O can be handled in a terminal, within the command line, or as part of a GUI
* See User I/O for more info

### Function and Method Calls

function\_name(argument)

function\_name(positional argument, parameter name=keyword argument)

function\_name(\*list or tuple argument, \*\*dictionary argument)

class.method()

* Function calls will always have parentheses at end, even if no arguments are being passed
* When calling a function, an argument must be supplied for all parameters that don’t have a default value
  + Arguments can be passed as positional arguments, where the order matches the arguments to the parameters, or as keyword arguments, where the parameters are included in the function call
    - Parameter names are treated as variables, not strings
  + When passing a list or tuple to a var-positional parameter, append an asterisk (\*) to the front of the variable name
    - This applies to both collections defined in the function call and variables of those collection types
  + When passing a dictionary to a var-keyword parameter, append a double asterisk (\*\*) to the front of the variable name
    - This applies to both dictionaries defined in the function call and dictionary variables
  + See Argument Syntax for more info
* See Calling Functions for more info
* Appendix C: Functions contains a list of methods and functions from Python, the standard library, and the modules listed in Imported Modules

### Mathematical Operations

int\_sum = int + int

float\_product = float \* int

* Mathematical operators
  + + Addition
  + - Subtraction
  + \* Multiplication
  + / Division
  + // Integer division
  + % Modulo (remainder)
  + \*\* Exponent
    - Number before operator is base; number after is exponent
* When performing math on numbers of two different types, the answer will be of the more precise type
  + When performing division, the answer will return a float, even if both the inputs are integers
* Precedence follows standard order of operations
  + Exponents come before unary negative operators, so to use a negative number as a base in an exponential operation, the unary negative and number need to be a parenthetical
* See Mathematical Operations for more math functions

### Unary Mathematical Operations

variable += number

variable -= number

rounding

* Mathematical operations using a single variable
  + - Negative number
* The += and -= operations increment or decrement variable by number
  + Most often used as a counter in loops where number is 1
* Making a number negative or positive comes after exponentiation in order of operations
* See Mathematical Operations for more math functions

### If Statement

if Boolean expression:

body of statement

* If Boolean expression evaluates to a value considered true, body of statement executes
  + See Boolean Values for info on what values are considered equal to True in Boolean expressions

if Boolean expression:

body of statement

elif Boolean expression:

body of statement

else:

body of statement

* elif allows for chaining multiple if statements together, while else creates a case for if none of the Boolean expressions are true
  + The conditional statements are evaluated in the order presented
  + Once a conditional statement evaluates to true, the code block below is executed and no other statements are evaluated
* See If Statements for more info

### While Loop

while Boolean expression:

body of loop

* While Boolean expression evaluates to true, body of loop is executed
  + The loop must contain an action to change variable in conditional statement to eventually make the Boolean expression false, otherwise the program will be stuck in the loop
* See While Loops for more info

### For Loop

for item in iterable:

body of loop

* For each item in iterable, body of loop is executed
  + item doesn’t need to be initialized before loop
  + iterable is either a collection object or a string
* h
* See For Loops for more info

### Errors

* Errors are raised when a problem emerges during the execution of the code
  + When an error is raised, execution of the program ends unless there’s error handling
* There are many different types of errors—a list is in Errors
* h

# Error handling for all errors

try:

code that might raise an error

except:

code to be executed if the above raises an error

* Above construct will perform the code block under try, and if an error is raised, the code block under except is executed
* The except block can be set up to be executed if only certain errors are raised—see Error Handling with Try for details
* h

### File I/O

open\_file = open(‘file path with file extension’, ‘mode’)

* Files are opened with the open function
  + The arguments are the file being opened—the name if the file and program are in the same directory, the file path if they’re not—and an optional mode
    - The values for mode are in Opening Files
* H
* See File I/O for more info

### Web I/O

Basic web I/O call

* Basic web i/o info
* See Web I/O for more info

### Database I/O

* Databases are idiosyncratic, so there’s no single syntax for utilizing the Python database API
* See Database I/O for more info

# Basic Data Types

## Numeric Data Types

* H

### Complex Numbers

* The imaginary number () is represented with the letter j
  + Appending a j to any other numeric data type makes it a complex number
  + There’s no built-in sorting for complex numbers

### Dates and Times

* The datetime module…

## Strings

* Strings are a built-in class
  + That class is an immutable ordered collection type made up of character-type objects
    - The character class is a data type containing one UTF-8 character
  + Because strings are immutable, identical strings have the same ID (memory address), even if initialized to different variables
  + The class has a large number of methods—see Strings for relevant functions and methods
  + Other classes can use the string class as a parent for inheritance
* String concatenation is done with a plus sign (+)
  + Two literal strings can also be concatenated by being put next to one another
* Strings can be made to repeat with an asterisk (\*)
  + The syntax is n \* string—this will repeat string n times
* Individual character(s) in a string can be referenced via Index Referencing and slicing

### Metacharacters

* Metacharacters are characters or combinations of characters that have special meaning when used in strings
* Metacharacters
  + \ Escape (make the next character literal)
  + \n Newline
  + \t Tab
* Raw strings treat all characters as literal, ignoring potential metacharacters
  + Raw strings are formed by putting r right before the opening quote character for the string

### String Formatting

f"{variable:formatting instructions} string”

* The format method used to format strings can be structured in two ways
  + “{} string {}”.format(x, y)
  + f”{x} string {y}”
* When using the standard dot notation, curly braces and method arguments can be matched by
  + Putting the arguments in order of use
  + Putting the argument position number (zero-indexed) inside the curly braces
  + Assigning keys to the arguments and putting the keys inside the curly braces
* When using f-strings (where the letter f immediately precedes the opening quotation marks), the case of the f doesn’t matter
* The curly braces represent the position of the variables being inserted into the string, but they can also contain formatting instructions after a colon (:) which is after any value-related content in the curly braces
  + <n: total string length n characters, value before colon left aligned, any additional needed characters are spaces
  + >n: total string length n characters, value before colon right aligned, any additional needed characters are spaces
  + ^n: total string length n characters, value before colon centered, any additional needed characters are spaces
    - For the items above, placing 0 before n will fill the empty spaces with 0
    - The components of the above can be used individually, but using them together is clearer
  + +: adds a plus sign before the characters
  + ,: uses comma as thousands separator
  + f: includes a default number of decimal places
  + .nf: includes n decimal places
  + x: present an integer value in hexadecimal with lowercase letters
  + b: presents an integer value in binary
  + n: presents a numeric value in base 10 with separators determined by location
  + %: presents a float as a percentage value, including the percent sign
  + e: presents a float in scientific notation with a default precision of six places using e to indicate the exponent
  + h
* After the variable name in the curly braces, encoding information can be added after an exclamation point (!)
  + a: ASCII
  + s: string
  + r: literal (like repr function)
* h

#### Using Dictionaries with String Formatting

dictionary = {key1: value1, key2: value2}

string = “{key1} string {key2}”.format(\*\*dictionary)

* Printing string will return value1 string value2
* The curly braces contain the keys of the dictionary that was passed into format as a var-keyword argument

### Regular Expressions

regex\_object = re.compile(r’regex’, flags)

match\_object = regex\_object.search(string)

find\_all\_matches = regex\_object.findall(string)

string\_matching\_regex = match\_object.group()

* Regular expressions, regexes for short, are symbol used to represent a pattern in text
* Regular expressions use the re module from the standard library—see Regular Expressions for formulas
* Matches with regexes are eager and greedy by default
  + Eager means the first possible match is returned
    - This applies to groups of characters separated by the or operator—the regex will try and match the groups in left to right order
  + Greedy means the longest possible string is matched
    - This applies to repetition in the regex—the repeating part will claim as many characters as possible before matching against the next part of the regex begins
    - In contrast, the lazy strategy matches as little as possible before moving on to the next part of the regex
* The regex module’s compile function creates a regex object for use in regex search functions
  + Saving the regex allows for it to be reused in multiple search functions
  + Using regex object provides natural grouping the group method can use
  + The regex is a raw string because otherwise a Python escape metacharacter would be required whenever a regex escape metacharacter was used
* Regex flags indicate different modes the regex can operate in
  + Multiple flags are allowed for the flags argument when separated with the bitwise or (|) character
  + Flag values exist for character encodings, but the need to change from the Unicode default is very rare
  + Flag values
    - re.IGNORECASE makes matching case insensitive
    - re.MULTILINE allows anchoring to beginning and end of lines
    - re.DOTALL includes newlines in the wildcard
    - re.VERBOSE ignores whitespace unless escaped/metacharacter or in a character set and allows for standard Python comments
      * These are generally triple quoted to take full advantage of the ignored whitespace
    - re.DEBUG lets the regex parser output debugging info
* The basic search function the first match to the pattern of regex\_object in string according to the search rules described above
  + The method for writing regexes to describe patterns found in text can be found in Constructing Regexes
  + The search function can contain the regex and flags as re.search(r’regex’, string, flags), but the regex object is easier
* Regex search functions return a match object if a match is found and None if there’s no match
  + Match objects have a Boolean equivalent of true, so if match object can be used to determine if a match was found
  + The group method can return the matched string for regexes completely enclosed in a group or used as regex objects
* The findall function searches for matches to regex\_pattern in string, but returns all matches
  + If regex\_pattern contains no groups, the matches will be returned as a list of strings
  + If there are groups, the matches are returned as a list of tuples, with each tuple representing a match and each string in the tuple representing a group
    - As a result, parts of the string matched to parts of the regex outside a group aren’t returned—to return the complete string, everything will need to be in a group
* Groups within a regex can be assigned names by placing ?P<name> at the very beginning of the group
  + The string matched by that group can be returned by the group method using ‘name’ as an argument

#### Constructing Regexes

* Regexes are constructed through a combination of literal characters and the Regex Metacharacters listed below
  + The syntax and metacharacters are different from Python
  + Regexes can be tested at [regex101.com](https://regex101.com/)
* Some metacharacters are contextual—some switch meanings or become literal characters depending on context
* Character categories
  + Word characters include all uppercase and lowercase letters, all single-digit numbers, and the underscore
  + Whitespace characters include the space, tab, newline, and carriage return
* Metacharacter categories
  + Grouping: allow a collection of characters to be treated as a unit
  + Set: create a group of characters, any one of which can be the character found
    - Combining negative character set shortcuts in a character set is not useful—the logic is “either not x or not y”, meaning that x will be selected by virtue of not being y
  + Repetition: the immediately preceding character, group, or set will repeat
    - None of these function as metacharacters within a character set
  + Anchor: these boundaries limit where the regex will match based on boundaries
    - These metacharacters are zero width—they don’t represent a character in the regex

##### Regex Metacharacters

* Basic metacharacters
  + . Wildcard (any single character except newline)
    - Not a metacharacter within a character set
  + \ Escape (treat next character as literal)
  + \t Tab
  + \n Newline
  + \r Carriage return (sometimes appears before on instead of newline)
* Grouping metacharacters
  + () Character group
    - Character sets can go within groups
    - $# is used to refer to the # group in the regex
  + | Or operator
    - Matches tried from left to right—eager means matching on first part of string will be tried against all options before moving on
    - Can be used outside a group, but much less useful
  + ?# Comment—a group that starts with this is a comment
* Set creation metacharacters
  + [] Character set
  + \d Digit character (numeric)
  + \w Word character
  + \s Whitespace character
  + \D Character other than digit (numeric) character
  + \W Character other than word character
  + \S Character other than whitespace character
* Set metacharacters—metacharacters used within sets
  + - Range of characters
  + ^ Negative character set (matches any single character other than those in the brackets)
    - Only a metacharacter when first in the set
* Repetition metacharacters
  + \* Preceding item repeats zero or more times
  + + Preceding item repeats one or more times
  + ? Preceding item is optional (repeats zero or one time)
  + {} Quantity of repetition of preceding item (when used as shown below)
    - {min,max} repeats between min and max times
    - {n} repeats n times
    - {min,} repeats at least min times
    - {,max} repeats no more than max times
    - If the regex string has the format method applied to it, these need to be double curly braces
  + m? Regex should be lazy when matching repetition represented by m repetition metacharacter
* Anchor metacharacters—reference position, not a character
  + ^ Start of string or line
  + $ End of string or line
  + \A Start of entire string
  + \Z End of entire string
  + \b Word boundary (before first word char in string, after last word char in string, between word char and non-word char)
  + \B Not a word boundary

## Collections

* Collections are classes where the objects contain other objects
  + For built-in collection classes, the other objects can be of any class and don’t need to be of the same class

### Ordered Collections

* Lists and tuples are the basis of all ordered collections—they’re very similar as both are
  + Sequences of data
  + Ordered with zero-based indexing
  + Iterable
* Primary difference is that lists are mutable—items can be added, removed, and rearranged—while tuples are immutable
* See Ordered Collections for related functions and methods

#### Index Referencing

* Elements in collections are referenced by variable[index], where variable is the name of the list or tuple and index is the zero-based position number
* Slices are groups of multiple elements from an index or list (is there a better definition?)
  + Including a colon (:) in index creates a slice
  + Slices can have up to three items structured as variable[start:stop:step]
    - start is the first element in the slice
    - stop is the first element after the end of (non-inclusive last element in) the slice
      * If stop is n, the last value in the slice is at index n-1
    - step is the interval at which elements are selected
    - Slices from the beginning to a certain index or from a certain index to the end can be constructed by not including an index number for start or stop respectively
  + Negative numbers in start or stop count from end; a negative number in step counts from start to stop
  + h

#### Tuples

tuple\_variable = (tuple element 1, tuple element 2)

tuple\_variable = tuple singleton,

tuple\_variable = tuple element 1, tuple element 2

unpack1, unpack2 = tuple\_variable

* Tuples with no items are initialized with empty parentheses; tuples with one item are initialized by adding a trailing comma to that item
* The creation of tuples without parentheses is called tuple packing
* Sequence unpacking involves putting a given number of variable names on the left of an assignment operator and a tuple with the same number of the right to assign the elements of the tuple to individual variables
  + Multiple assignment is sequence unpacking and tuple packing in a single statement

#### Lists

list\_variable = [list element 1, list element 2]

list\_variable = list(list element 1, list element 2)

* Lists can be concatenated with the plus sign (+)
* Individual objects in the list can be changed by using Index Referencing (list[index]) followed by the assignment operator
* See List Manipulation for relevant functions and methods

##### List Comprehension

[item <transformation> for item in list]

[item <transformation> for item in list if Boolean expression]

[(item <transformation1>, item <transformation2>) for item in list]

{item: item <transformation> for item in list}

* A list comprehension is a collection based off another list or an iterator
  + The collection will be initialized as a list or dictionary
* <transformation> is a stand-in for any type of change—it can be:
  + A mathematical operation
  + A string manipulation
  + Or anything other expression that can be enacted on all potential values of item, including another list comprehension
  + This is a common place for Lambda Expressions
* Just like with a standard for loop, item and list can be initialized in the list comprehension
* Adding if Boolean expression to the end will return a list with only those items where the expression is true
* item <transformation> can be a tuple with multiple instances of item <transformation>, which will return a tuple for each instance of item with values corresponding to those in the comprehension’s tuple
* What the list comprehension is wrapped in will determine the type it is returned as
  + For the dictionary, either the key or the value can be the comprehension as long as the keys in the created dictionary will be unique
* Generator expressions have the same syntax as list comprehensions but are enclosed in parentheses and are used within function calls

#### Stacks

* Stacks are lists where the elements most recently added is the first element retrieved—“last in, first out”
  + Elements are added to with stack.append()
  + Elements are removed with stack.pop()

#### Queues

from collections import deque

queue.append(element being added to queue)

queue.popleft()

* Queues are lists where the first element added is the first element retrieved—“first in, first out”
* Queues should be implemented with the collections.deque class
  + See Queues and Deques for more details on the methods

#### Deques

* Deques are lists where elements can be added or removed from the beginning or end
* Queues should be implemented with the collections.deque class
  + See Queues and Deques for the methods

#### Multidimensional Array

* List of lists
* Discuss when to use vs. using dataframe, link to section on dataframe

### Dictionaries

dictionary\_variable = {key1: value1, key2: value2}

dictionary\_variable = dict(key1 = value1, key2 = value2)

dictionary\_variable = dict([(key1, value1), (key2, value2)])

* Dictionaries are unordered collections where the elements are indexed by keys
  + Functionally, this makes the key-value pairs the elements in dictionaries
* Values are referenced by dictionary[key]
  + The above indexing value can be used in assignment statement to replace an existing value or add a new value to dictionary
  + To determine if a value is used as a key in a dictionary, use the keyword in
  + Attempts to access a key that doesn’t exist will return a KeyError
* Keys must be an immutable type and unique
  + Tuples can be keys if they contain only immutable objects
* When using keyword arguments in the constructor method (second example), string keys don’t need to be enclosed in quotes
* See Dictionary Manipulation for dictionary class methods
  + Dictionary Views are special methods that create iterators from the dictionary

### Sets

set\_variable = {set element 1, set element 2}

set\_variable = set(element)

* Sets are unordered collections of unique items
* Sets can be initialized with element(s) not in compliance with the rules of sets, but the variable will be saved in compliance with the rules
  + For example, saving the string “hello” as a set will result in a set containing {h, e, l, o}
* While sets are designated with curly braces, initializing a variable with empty curly braces creates an empty dictionary; to make an empty set, use the constructor method
* Set comprehensions can be created with the same syntax as List Comprehensions

#### Membership Operations

* Sets are primarily used for comparing the membership of sets
* Membership comparison operations
  + x & y And: in x and in y
  + x | y Or: in x or in y
  + x ^ y Xor: in x or in y but not in x and in y
  + x – y Not: in x but not in y

# Control Flow Statements

* Control flow is the order in which statements are executed; control flow statements are statements which initiate a choice about executing other statements
  + Computer choices are always binary—yes or no

## Boolean Expressions

* Boolean expressions compare two values, evaluating down to a Boolean value

### Boolean Values

* There are only two Boolean values: True and False
  + Both words beginning with a capital letter are keywords
* For the purpose of evaluation and comparison, the values below are considered False
  + Numeric data types with value 0
  + Empty strings
  + None
* Correspondingly, all values other than the ones above correspond to True
* h

### Boolean Operators

* Boolean expressions utilize operators for analyzing the relationship between two values
  + The exception is the use of not as a unary operator, which checks for inversion
* Boolean operators by category
  + Comparison operators—compare two values
    - == equals
    - != not equal
    - < less than
    - > greater than
    - <= less than or equal to
    - >= greater than or equal to
    - Multiple comparisons can be used in a single expression
  + Identity operators—compare the memory address (ID) of two values (confirm this is accurate)
    - is the same ID
    - is not different IDs
  + Membership operators—compare a value to the members of a collection
    - in value is in the collection
    - not in value isn’t in the collection
    - The single value goes before the operator, the collection goes after the operator
  + Logical operators—get a definition for these, confirm the definitions below and remove parentheticals
    - and both values (x and y)
    - or either value (x or y)
    - not negates a value or the outcome of an expression

## If Statements

if Boolean expression:

body of statement

elif Boolean expression:

body of statement

else:

body of statement

* Construction “if variable in tuple” can be used to have multiple possible values for variable execute same code block—all of the possible variables go into tuple, which can just exist in Boolean expression eg. If answer in (‘yes’,’y’): --this needs to be better/more accurately phrased using the type of operator “in” is
* g

### Ternary Operator

bool\_variable = evaluates to a Boolean value

variable = true condition if bool\_variable else false condition

* If-else on single line
* Complete statement required to work, but either condition can return None

## While Loops

while Boolean expression:

body of loop

* H

## For Loops

for item in iterable:

body of loop

* Attempting to modify what objects are or are not in a collection while iterating though it is not recommended—creating a new collection is a better option
* To iterate over a sequence of numbers, use the range function—see Control Flow Statements for more info

### Dictionary Views

for k, v in dictionary.items():

body of loop

for k in dictionary.keys():

body of loop

for v in dictionary.values():

body of loop

* Dictionary object has iterator methods
  + dictionary.keys() iterates over keys
  + dictionary.values() iterates over values
  + dictionary.items() iterates over both keys and values
* Use of k and v as variables for keys and variables respectively within loop isn’t required but is traditional

## Control Flow Keywords

* continue: returns to top of code block to check Boolean expression again
* break: exits code block
* else: executes the code block that follows if a loop exits normally (not via break) or if an if-elif chain reaches it
  + For for loops, finishing iterable; for while loops, Boolean expression becomes a value considered false
  + Placed at same level of indentation as line starting control flow statement
  + Also used with the try keyword—see Error Handling with Try for more info
* pass: does nothing
  + Used when something is required for syntax but not for functionality
  + Can also serve as a placeholder for the code in function and class definitions
* finally: indicates code to be executed just before a continue or break statement executes
* h

# Object Model Extensibility and Reusability

Overview of python’s object model???

Everything is an object and objects are instances of classes

Is this the place for infor on object-oriented programming?

## Functions and Methods

* Basic information on creating functions is in Function Definitions; basic information on calling functions is in Function and Method Calls
* Functions and Python’s object model
  + Everything in Python is an object, so functions without the parentheses at the end can be called as objects

### Namespaces and Variable Scope

* When using variables for default values for arguments, the value used is based on the variable value at the point in the program where the function is defined
* Namespaces are formally defined as mappings from names to objects
  + Objects in different namespaces have no relation to one another—two objects with same name from different namespaces are unrelated
  + The namespace of an object is specified through the dot notation construction namespace.object
  + The top-level namespace for a program is called \_\_main\_\_ (double underscores on each side)
* A scope is a part of a program where a namespace’s variables can be used without referencing the namespace via dot notation[[1]](#footnote-1)
  + Scopes are searched in this order
    - Local names/namespace
    - Enclosing functions—searched from most recently called out
    - Current module (\_\_main\_\_)
    - Built-in Python namespaces
  + Variables are assigned to the innermost scope

#### Declaring Global Variables

global variable

* The keyword global puts the following variable in the \_\_main\_\_ namespace—the scope of the current module—instead of the innermost scope
  + Multiple variables can be declared on one line with global; multiple assignment may also be possible

#### Call-By-Value and Call-By-Reference

* H
* How these relates to scope and mutability of objects
* How changes in variable values are determined by the call-by-value model (see Python essential training 7.2 for a little info)

### Parameter Syntax

def function(parameter1, parameter2 = default parameter2 value):

def function(\*var-positional parameter, \*\*var-keyword parameter):

def function(positional-only, /, positional-or-keyword, \*, keyword-only):

* There are five types of parameters
  + Positional-or-keyword parameters can take values from positional or keyword arguments
    - This is the default parameter type
  + Positional-only parameters take values only from positional arguments
  + Keyword-only parameters take values only from keyword arguments
  + Var-positional parameters take a collection of values as a positional argument
    - Effectively, this allows a function to take in a varying amount of input by grouping that input into a collection and passing the collection to a parameter
  + Var-keyword parameters take a collection of values as a keyword argument
    - Essentially, this allows for a dictionary as an argument
* Var-positional and var-keyword parameter names must be preceded by one (\*) or two (\*\*) asterisks respectively
  + The use of variable names args and kwargs respectively is traditional but not required
* Positional-only and keyword-only parameters are designated by adding a slash (/) or an asterisk (\*) to the list of parameters
  + Positional-only parameters go before the slash in the parameter list
    - If the last item in the parameter list is a slash, all the parameters will be positional-only
  + Keyword-only parameters go after the asterisk in the parameter list
    - If the first item in the parameter list is an asterisk, all the parameters will be keyword-only
* Any parameters going after a var-positional parameter should be explicitly keyword-only—they’re implicitly that way as not using the keyword will make them part of the collection

### Generators

def generator\_name(parameter1, parameter2):

“”” docstring “””

body of generator

loop:

yield return value

* Generators are a special kind of function that return multiple values
  + Keyword yield indicates return value, but function continues after hitting that keyword
  + yield need not be in a loop, but must useful in that situation
* Generators do the same thing as Creating Iterators, but the special methods are created automatically, making generators much more compact

### Decorators

def function\_name (parameter):

body of function

returnreturn value

@

Decorator

function

* A decorator function returns a wrapper function
* Don’t quite understand these from the video—do more research

### Calling Functions

#### Methods and Dot Notation

* Methods are called using dot notation, structured as object.method()
  + object must be of the type of class method belongs to
* The rules for methods are the same for classes built in to Python, classes in imported modules, and created classes

#### Argument Syntax

function(positional argument, parameter name=keyword argument)

function(\*\*{‘parameter name’: keyword argument})

function(\*\*dictionary variable)

function(\*tuple or list variable)

function(positional argument, parameter name=keyword argument)

* There are two types of arguments—positional and keyword—for determining how the argument is matched to a parameter in the function
  + Positional arguments are matched based on being in the same index in the function call and the opening of the function definition
    - Formally, positional arguments are arguments that aren’t keyword arguments
  + Keyword arguments use the parameter name as a key and the argument as a value
    - This can be done through assignment within the call, the creation of a dictionary in the call, or the use of a dictionary variable as an argument
    - When using a dictionary as a variable or constructed in the call, it needs to be preceded by two asterisks (\*\*)
* The var-positional and var-keyword parameters allow for collections to be taken as an argument
  + Var-positional parameters take a collection of values as a positional argument, so the argument is a collection with zero-based indexing
    - The list or tuple can be created in the function call or be a variable, but it needs to be preceded by an asterisk (\*)
  + Var-keyword parameters take a collection of values as a keyword argument, so the argument is a dictionary (the only collection type with keyword indexing)
    - The same rules as with keyword arguments apply

### Lambda Expressions

lambda varable1, variable2: expression with variable1 and variable2

* The lambda keyword allows for the creation of short, single-use functions

## Classes and Object-Oriented Programming

* Classes are the basis of object-oriented programming
* Classes defined within a program appear at the beginning and are instantiated later
  + Methods are called via dot notation—see Methods and Dot Notation
    - Class methods can be used without instantiating an object—the class can be called within the method call in the form method\_name(class\_name())
* Classes serve as their own namespaces

### Defining a Class

class class\_name:

“””Intro to the class.

More details about the class.

“””

# Constructor Method

def\_\_init\_\_(self, parameter):

self.\_attribute = parameter

# Getter/Setter Methods

def attribute(self, NewValue = None):

if NewValue: return self.\_attribute = NewValue

return self.\_attribute

**# Other Methods**

def method1(self):

body of function

def method2(self, parameter corresponding to an argument):

body of function

* Class definitions can be roughly broken up into three parts
  + Constructor method: \_\_init\_\_
    - The constructor method allows for the attributes of newly created objects to be set to a custom initial state through requiring arguments at instantiation and/or default values
      * The parameters indicate the arguments needed when instantiating the class
      * Each attribute has an assignment operation
      * The attribute variable names begin with an underscore to indicate that they should be treated as private
        + It also often has the same text name as the parameter
  + Getter/setter methods
    - A variant on the getter method that can also be used to change the value of the attribute
      * If the method has no argument (other than the object), the default value means the if statement is skipped and the current attribute value is returned
      * If the method has an argument in the parentheses, the if statement evaluates to true, assigning the value of the argument to the attribute then returning the attribute
        + This won’t work if the attribute is being set to something that evaluates to False—those values are listed in Boolean Expressions
  + Other methods
    - These are the other functions specific to the class
    - They’re called in the function using the format described in Methods and Dot Notation
      * The above format actually is a shorthand for class\_name.method(object)
* Other special methods used for defining classes include
  + \_\_str\_\_: provides string representation of object
  + \_\_iter\_\_: identifies class as an iterator
  + \_\_next\_\_: for use in iterators, indicates the iteration should repeat
  + The special class methods have double underscores before and after the name
* The docstring format above allows for the docstring to be called via the \_\_doc\_\_ attribute

#### Using a Constructor Method with Var-Keyword Parameters

class class\_name:

def\_\_init\_\_(self, \*\*kwargs):

self.\_attribute1 = kwargs[argument key 1]

self.\_attribute2 = kwargs[argument key 2]

self.\_attribute3 = kwargs[argument key 3]

self.\_atrb4 = kwargs[key] if key in kwargs else default

**# Creating an instance of class\_name**

instance =class\_name(

argument key 1=value of attribute 1,

argument key 2=value of attribute 2,

argument key 3=value of attribute 3

**)**

* Using \*\*kwargs in \_\_init\_\_ allows for all argument(s) at point of object initialization to be keyword argument(s)
  + Allows the attributes to be called in any order
  + Allows for attributes to have default values with a Ternary Operator
* H
* See Parameter Syntax for info on var-keyword parameters and Argument Syntax for info on keyword arguments

#### Class Variables

class class\_name:

class\_variable = value shared by all objects of this class type

* Variables initialized in a class definition but defined outside the constructor method are shared by all objects
  + Unlike attributes, class variables belong to the class, not the objects of type class
    - All objects of the class type have this attribute, but it’s the same value for all objects
  + Class variables are rarely used

#### Creating Iterators

class iterator\_class:

def \_\_init\_\_(self, iterable, other parameters):

definitions for all parameters, including iterable

self.index = len(iterable)

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.index == 0:

raise StopIteration

self.index = self.index – 1

desired actions to be taken on each item in iterable

self.iterable[self.index] is current item in iterable

* An iterator takes in iterable—an ordered collection—and performs a given action upon each item in the collection
* The iteration is executed with a for loop where the collection being iterated over is an object of type iterator\_class

#### Creating String Representations

h

### Attributes

* Attributes and methods make up a class, but attributes don’t appear directly in the class definitions—they’re represented by the constructor and getter/setter methods, which are used to initialize, call, and change attribute values
  + Attributes can be referenced directly via dot notation, but using getter/setter methods is best practice
    - The underscores in front of attribute names are a Python convention to signal that the variables should be treated as private, as Python has no private variables

### Inheritance

class child class name(parent class):

* Inheritance allows a class to use the attributes and methods of another class
* The parent class can be from another module, in which case the module name is appended to the front of parent class via dot notation
* A class can have multiple parent classes
  + Methods are searched for in the parent classes from left to right in the arguments list, with the all nested functions and inherited classes being searched before moving on to the next class

#### Overriding

* A child class can override methods from a parent class by including methods with the same name as those in the parent class
* An overridden class from a parent can still be used by calling it as parent\_class.method(object, arguments) so long as parent\_class is in the global scope

#### Overloading

* Put info here

## Modules and Packages

* Modules allow for extensive code sharing and reusability in addition to code organization
  + Packages allow for even greater convenience by allowing reusable code to be spread across multiple files but kept together
* Python files
  + Python code is saved in plain text files with the file extension .py
  + Within a file, the file name as a string is accessed with the attribute \_\_name\_\_ (double underscores on each side)
* The content in a Python file can be used in another Python file through Import Statements
  + Thanks to Python’s object model, a function from an imported module can be given a local name by putting the complete function name without parentheses after an assignment operator
* h

### Python Standard Library

* The Python Standard Library is a collection of modules and packages that are automatically installed with the standard Python distribution
* H
* The list of Python Standard Library modules integrated into this documentation is in Python Standard Library

### Package Installation and pip

* Beyond the Python Standard Library, there are a wide variety of preexisting modules available for use
  + Third-party modules are indexed in the Python Package Index (PyPi)
* PyPi modules must be installed before use; in the standard Python distribution, modules are installed via pip

#### Using pip

* The ???, aka pip, is the command line tool for installing PyPi modules
* The current version of a module can be installed by entering pip install module
  + Specific versions can be installed by adding ==version number at the end of the command
  + Already installed modules can be upgraded with the command pip install --upgrade module
  + A file can be generated to load Python modules
    - A text file listing all the installed modules is generated with pip freeze > requirements.txt
    - The file can be used for installing all the modules listed within with pip install -r requirements.txt
* Packages can be uninstalled with pip uninstall module
* pip list displays a list of the installed packages

### Custom Modules

* Python files can be imported into other Python files as modules
* Compiled versions of Python files imported into other Python files are saved in a file named \_\_pycache\_\_ with a file extension of the Python version and .pyc
  + Python will automatically check the modification date of the .py version against that of the .pyc file and recompile the latter if necessary

# Input/Output

## User I/O

* Data to or from user
* Terminal, command line, GUI
* See User I/O for general user I/O functions and methods
* Validation module

## File I/O

* Python can read data from and write data to a variety of file types, which can generally be divided into two categories
  + Text files, which only contain text
  + Binary files, which contain some form of embedded data

### Opening Files

open\_file = open(‘file path with file extension’, ‘mode’)

* The open function returns a fileIO object, which is an iterator
  + The fileIO object created has a position in the file
    - The object’s position is used to determine where reads and writes from the file begin
    - When the object is created, it’s positioned at the beginning of the file
      * The object’s position can be returned and changed with methods
  + File object methods are in File I/O
* Open mode options
  + r read only (default)
  + w write mode—new content replaces existing content
  + a append mode—adds new content after existing content
  + Some extra info
    - Write and append modes will create the file if it doesn’t exist
    - Adding a plus (+) after the letter will allow read and write access
    - Adding b or t after the open mode option can specify if the file should be opened as binary or text (text is default)

### Reading from Files

* The two primary methods for reading files are fileIO.read() and fileIO.readline()
  + fileIO.read() returns the complete file or, if given an argument, the given number of characters (if text) or bytes (if binary)
  + fileIO.readline() returns the file through the next newline character (\n)
* When the fileIO object reaches the end of a file, read methods return an empty string

#### Reading Text Files

for line in fileIO for text file:

method for outputting the lines of the file

* Text files are often read line by line with a for loop
  + Using line as the item name variable isn’t required, but it’s common as it’s a literal description
  + line is also a fileIO object and is used with any fileIO methods in the loop
  + See Reading Files for more info

#### Reading Binary Files

while True:

buffer =fileIO for binary file.read(byte size**)**

**if buffer: # If there’s no more file to read, will return False**

method for outputting part of the file

**else: break**

* The nested loops create a construct that will continue to read in the file in byte size pieces until there’s nothing left, at which point the loops end
* byte size should be a power of two

#### Reading JSON Files

import json

JSON\_file\_to\_Python\_object = json.load(fileIO)

JSON\_formatted\_string\_to\_Python\_object = json.loads(string)

* It’s possible to turn a JSON into a Python object with the JSON standard library module
  + This process is called deserializing and if formally defined as reconstructing the data from the string representation
* The parse\_float and parse\_int keyword arguments can accept the name of any numeric data type, not just the ones built in to Python (e.g. decimal.Decimal)
* See Reading JSON for more info on relevant methods

### Writing to Files

* H

#### Writing Text Files

* H

#### Writing Binary Files

* H

#### Writing JSON Files

import json

Python\_object\_to\_JSON\_file = json.dump(object, fileIO)

Python\_object\_to\_JSON\_formatted\_string = json.dumps(object)

* It’s possible to turn a Python dictionary into a JSON object with the JSON standard library module
  + This process is called serializing and is formally defined as converting Python data hierarchies to string representations
    - Realistically, the object being converted should be a dictionary, as anything else will not be a complete JSON
* Keys in JSON are always strings, so any keys of other types will be converted to strings during the write process
* indent keyword argument values
  + None: the most compact presentation, so no pretty printing (default)
  + Negative number, 0, or empty string: pretty printing with newlines but no indentation
  + Positive number: pretty printing will use that number of spaces for indentation
    - The number can be an integer or string
    - A string value can also be used for indentation, including meaningful escaped characters (e.g. \t)
* See Writing JSON for more info on relevant methods

### Closing Files

file\_name.close()

* At the end of writing to files, use the close method to close the file—otherwise, the data may not actually be saved to the file
* Calling a method on a closed fileIO object will result in an error

## Web I/O

* H

## Database I/O

Import database type

Database variable = database type.connect(‘name of database file’)

Cursor variable = database variable.cursor()

Cursor variable.execute(“””

SQL goes here

“””)

* Python’s database API provides a way to use Python to connect with databases, including sending SQL as output as reading the results of SQL select statements as input
* Because database systems are moderately idiosyncratic, somewhat difficult to generalize about usage of different databases with the API
* https://realpython.com/python-sql-libraries/
* This section needs much more work

### Maria DB I/O

* Work on this section because this is what will be needed for CORAL

### MySQL I/O

* hhh

### SQLite I/O

* Work on this section because this is likely to be what gets used by pySUSHI implementation

# Dataframes

h

# GUI Creation

## Creating the GUI Class

class GUI:

“””docstring”””

def \_\_init\_\_(self, root):

self.variable = ttk.widget(root, arguments)

self.variable.geometry(arguments)

ttk.widget(root, arguments).geometry(arguments)

def method(self):

some code involving changes to self.variable

* All of the widgets are created in the constructor method and all of the handler methods are in the class as other methods
  + Unlike a standard class, there aren’t getter/setter methods
* Constructor method
  + The sole parameter is root, the root window for the GUI
  + Assignment to self.variable is necessary only for those widgets that will be referenced in the handler methods
    - For those widgets not assigned to a variable, immediate placement via a geometry manager is necessary
  + Geometry management for widgets is performed using geometry methods
    - The syntax for using geometry management method with arguments is .geometry(arguments)
* Other methods—handler methods

## Widgets

* Using themed widgets requires the import statement from tkinter import ttk in addition to the basic tkinter import statement
* Getting and setting widget attributes
  + Indexing a widget on an attribute uses the syntax widget[‘attribute’]
    - The indexed widget alone will return the current value for the attribute
    - Setting the indexed widget to the left of an assignment operator allows for the attribute to be changed to the value on the right of the operator
  + The config method can change multiple attributes at once by taking the attributes as keyword arguments with the new values as the argument values
    - Calling config without arguments will return a dictionary of the widget’s attributes where the possible values for the attribute as well as the current value are combined in a tuple

### Labels

G

g

### Buttons

G

g

## Geometry Management

### Pack Geometry Manager

H

### Grid Geometry Manager

H

### Place Geometry Manager

h

## Event Handling

h

# Functionality Aspects

## Errors

* Errors are raised when there’s a problem with the execution of the code
  + Errors cause execution to stop with traceback followed by a line containing the error type and details on the error
  + Divided into syntax errors and exceptions
    - Syntax errors are problems with how the code is written
    - Exceptions are run-time errors
* The different types of built-in exceptions (called exception classes) include
  + KeyError: referencing a key that doesn’t exist in that dictionary
  + TypeError: the object in an operation or function is the wrong type
  + ValueError: the object in an operation or function is an invalid value but is the right type
  + ZeroDivisionError: dividing by zero
  + hh
* Info about an error can be provided by sys.exc\_info()
* H

### Error Handling with Try Statements

try:

code that might raise an error

except TypeOfError:

code to be executed if the above raises a TypeOfError error

**else:**

code to be executed if the first block doesn’t raise an error

**try:**

code that might raise an error

**finally:**

code to be executed just before the try block completes

* Error handling provides a way for a program to continue execution even if an error is encountered
* The above shows how to handle a specific error type; using just the except keyword will catch all keywords
  + Multiple except statements for different types of errors can be added in sequence
  + Multiple types of exceptions can be handled by a single except statement by listing the exception types in a tuple
  + A generic except can be put after type-specific except statements
* Including the raise keyword at the end of the except block will execute that block of code then raise the error in the standard Python fashion
* A finally clause will run just before a try statement completes regardless of the that statement’s result
  + If an error is raised, it happens after the finally block completes

### Raising Errors in Functions and Classes

def function\_name (parameter):

if Boolean expression returning True if parameter is valid:

body of function

**else:**

**raise** TypeOfError(error message)

* The raise keyword allows for indicating where an error should be raised in a custom function or class
* TypeOfError can be any of the listed error types
* If there’s no custom error message, parentheses aren’t needed after TypeOfError
* These are almost always nested in an if loop testing for the condition prompting the error
* It’s possible to create custom exception classes
  + Custom error classes inherit from a base error class that inherits from the class Exception
  + The classes usually contain just an init method for information about the error

## Sorting

### Making Comparisons

* Objects of types that aren’t comparable can’t be sorted together
  + Strings and numeric types can’t be compared
  + None can’t be compared with any other object
  + Complex numbers don’t have a defined method for comparisons
  + Python will attempt implicit conversions to try and make objects comparable
* Collections are compared using lexicographical ordering
  + The first items are compared
    - If different, the sequences are ordered based on the order of those two elements
    - If the same, the items at the next index position are compared
  + The above repeats until different items are found or one of the collections ends, at which point the smaller collection is confirmed as smaller in the comparison

### Basic Sorting Functions

* Python sorting functions put collections in ascending order based on value (if numeric) or Unicode character code (if non-numeric)
  + Strings sort on the character level unless a split method designating another delimiter is used on the argument
* Sorts display sort stability—when equal values are sorted, they remain in the same order as in the original collection
* The sorted function
  + A built-in Python function that works on all ordered collections, sets, and strings
  + The return value is an ordered list, regardless of the type of the original collection
    - The object used as an argument is not changed in any way
  + See Iterators Based on Ordered Collections for info on the function
* The list sort method
  + A list method that sorts the list in place with no return value
  + See List Manipulation for info on the method
* For both of the above functions, the keyword argument key allows for sorting based on a function
  + The function used must take one argument, and all the items in the collection being sorted must be able to arguments
    - Lambda Expressions can also be used
  + The key argument also allows for sorting objects of a custom class type—a lambda expression can specify the attribute to be sorted on
    - The syntax of the argument would be key=lambda x: getattr(x, “attribute name”)

## Docstrings

def function():

“””This is a summary of the function.

Here’s more information about the function.

“””

* Docstrings (documentation strings) provide information about a function or class
* Starting the docstring at the first line of the code block allows it to be called by the \_\_doc\_\_ attribute
  + Both functions and classes have it
  + As an attribute, it uses dot notation but no parentheses
* Docstring format
  + Right after the opening punctuation is a complete sentence summary of the class or function
  + If more lines are needed, a blank line goes between the first line and the rest of the docstring
  + If the docstring spans multiple lines, the closing punctuation is on its own line

## Code Style and Formatting

* Four space indentation
* Use blank lines to separate functions and classes and to seperate chunks within functions
* Use spaces around operators and after commas
* Name consistently, and use different conventions for classes vs. functions/methods
  + Convention is
    - PascalCase for classes
    - lowercase\_with\_underscores for functions and methods

## Bitwise Operations

Used for comparing bit format variables

* & and
* | or
* ^ xor (one or the other, but not both)
* << shift left

>> shift right

# Appendices

## Appendix A: Glossary

* Argument: a value passed to a function in a function call
* Attribute: a value associated with an object; any direct reference to it utilizes dot notation
* Class constructors: a special group of functions that instantiate their argument as an object of the given class if possible; because they are structured and used like functions, they’re listed in Appendix C: Functions and Methods
* Class: a template for objects created by a user; see Classes and Object-Oriented Programming for info on creating classes
* Function: a block of code that can accept values (arguments) and always returns values
* Method: a function defined inside a class
* Module: a group of classes, functions, and/or objects in a single file; a namespace serving as an organizational unit of Python code, see Modules and Packages for more info
* Object: any individual element of data with attributes and methods; something that can be a Python variable
* Parameter: a variable declared in a function definition that specifies the value(s) (argument(s)) the function accepts, the parameter name(s) are also used to represent the value(s) in the body of the function
* Script: a file containing Python code intended for execution
* Package: a collection of modules

## Appendix B: Python Modules

This appendix lists all the modules with functionality explicitly handled in this documentation. The import statement column includes the most common way to import the module into a program, including common aliases.

### Python Standard Library

|  |  |  |
| --- | --- | --- |
| Module | Import Statement | Description |
| builtins | import builtins | Provides access to Python’s built-in objects |
| collections | from collections import class | Provides queue and deque structure |
| datetime | From datetime import class | Working with dates and times |
| decimal | from decimal import \* | Accurate decimal calculations |
| importlib |  | Supports the implementation of import statements |
| io |  |  |
| json | import json | Serialize and deserialize JSON |
| logging | import logging | Creates logging system |
| math | import math | Floating point math (not anywhere else) |
| os | import os | Interaction with the OS |
| pprint | import pprint | Enables formatted output (not anywhere else) |
| random | import random | Making random selections |
| re | import re | Regular expression processing |
| statistics | import statistics | Calculate basic statistics (not anywhere else) |
| sys | import sys |  |
| textwrap | import textwrap | Paragraph formatting (not anywhere else) |

### Imported Modules

|  |  |  |
| --- | --- | --- |
| Module | Import Statement | Description |
| openpyxl | import openpyxl | d |
| tkinter | from tkinter import \* | Create GUI |
| c |  |  |
| c |  |  |
| c |  |  |

## Appendix C: Functions and Methods

This is a listing of useful functions and methods in Python, both built in and in the modules listed in Appendix B: Python Modules. For methods that act upon a specific object of a class, the name of the class will appear as varying Python code where the object itself should go. Arguments presented as keyword arguments must be used as such when the function is called.

### User I/O

|  |
| --- |
| input(argument) |
| Outputs argument to the terminal, then waits for user input; that input up to newline character is captured by the program and can be saved to a variable |

|  |  |  |  |
| --- | --- | --- | --- |
| print(argument, sep=sep, end=end, file=file, flush=flush) | | | |
| Outputs argument to file | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| argument | The printed string; multiple positional inputs can be given for this argument | No |  |
| sep | Character separating the multiple arguments given for argument | Yes | Space |
| end | Character at the end of the output | Yes | Newline (\n) |
| file | Output location | Yes | Terminal (sys.stdout) |
| flush | I/O buffer is flushed after execution | Yes | False |

|  |
| --- |
| chr(argument) |
| Returns the image corresponding to Unicode number argument |

|  |
| --- |
| ord(‘argument’) |
| Returns the Unicode number of the image in position argument |

### Mathematical Operations

See Mathematical Operations and Unary Mathematical Operations for basic operators

|  |
| --- |
| int(x) |
| Integer class constructor |

|  |
| --- |
| float(x) |
| Float class constructor |

|  |
| --- |
| complex(x) |
| Complex number class constructor |

|  |
| --- |
| + |
| ??? function as unary operator (will not make a negative number into a positive number) |

|  |
| --- |
| abs(x) |
| Absolute value of x |

|  |
| --- |
| divmod(x, y) |
| Calculates x/y, then returns (modulo, remainder) tuple as result |

Dates and Times

|  |  |
| --- | --- |
| datetime.datetime.now() | |
| Returns the current date and time | |
| **Note** | The returned value has attributes year, month, day, hour, minute, second, and microsecond that can be called with dot notation |

### Strings

|  |
| --- |
| str(argument) |
| String class constructor |

|  |
| --- |
| repr(string) |
| Returns the literal representation of string, including formatting as escaped characters |

|  |  |
| --- | --- |
| “string {}”.format(argument) | |
| Replaces the curly braces ({}) in the string before the dot operator with argument | |
| **Note** | For more info on this function, see String Formatting |

|  |
| --- |
| string.join(collection) |
| Creates string by combining all elements of the list or tuple collection separated by string |

|  |  |  |  |
| --- | --- | --- | --- |
| string.split(argument) | | | |
| Makes string into a list where the elements are created by splitting string at each instance of argument | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| argument | Delimiter for splitting string | Yes | Whitespace characters |
| **Note** | Delimiter is not preserved | | |

|  |
| --- |
| string.rstrip() |
| Strips all whitespace characters from the end of string |

#### String Case Changes

|  |
| --- |
| string.upper() |
| Makes all letters in string uppercase |

|  |
| --- |
| string.lower() |
| Makes all letters in string lowercase |

|  |
| --- |
| string.capitalize() |
| Capitalizes the first letter in string |

|  |
| --- |
| string.swapcase() |
| Changes the case of all letters in string |

|  |
| --- |
| string.title() |
| Makes the first letter of each word in string uppercase and the rest lowercase |

|  |
| --- |
| string.casefold() |
| Makes all letters in string lowercase even in unicode |

#### Regular Expressions

|  |  |  |  |
| --- | --- | --- | --- |
| re.compile(regex, flags) | | | |
| Returns a regex object allowing for regex to be used in regex module search methods | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| regex | Regex built according to Constructing Regexes | No |  |
| flags | Changes to the way the search is conducted | Yes | None |
| **Note** | See Regular Expressions for info on how to use, including possible values for flags | | |

|  |  |  |  |
| --- | --- | --- | --- |
| re.search(regex, string, flags) | | | |
| Returns the match object for the first match to regex in string | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| regex | Regex built according to Constructing Regexes | No |  |
| string | The string being searched | No |  |
| flags | Changes to the way the search is conducted | Yes | None |
| **Note** | See Regular Expressions for info on how to use, including possible values for flags; the method can also be called on a regex object, taking only the argument string | | |

|  |  |  |  |
| --- | --- | --- | --- |
| re.findall(regex, string, flags) | | | |
| Returns the match object for all of the matches to regex in string | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| regex | Regex built according to Constructing Regexes | No |  |
| string | The string being searched | No |  |
| flags | Changes to the way the search is conducted | Yes | None |
| **Note** | See Regular Expressions for info on how to use, including possible values for flags; the method can also be called on a regex object, taking only the argument string | | |

|  |  |  |  |
| --- | --- | --- | --- |
| match\_object.group(argument) | | | |
| Returns the strings matched to all the groups in match\_object | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| argument | The number or name of the group to be returned | Yes | Returns everything matched to grouped content |
| **Note** | By grouping an entire regex, this can be used to return the match; counting groups starts at one; for more info, see Regular Expressions | | |

### Ordered Collections

For all instances of index, slicing can be used unless otherwise indicated; see Index Referencing for info on slicing.

|  |
| --- |
| list(x) |
| List class constructor |

|  |
| --- |
| sum(collection, argument) |
| Returns the sum of the elements in collection; if argument is included in the call, it will also be included in the returned sum |

|  |  |  |  |
| --- | --- | --- | --- |
| collection.index(argument, start, end) | | | |
| Returns the index of argument in collection | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| argument | The item being searched for | No |  |
| start | The index to start searching at | Yes | 0 |
| end | The index to stop searching at | Yes | Index of last item in collection |

|  |
| --- |
| collection.count(argument) |
| Returns the number of times argument appears in collection |

#### Ordered Collection Properties

|  |
| --- |
| len(collection) |
| Returns the number of elements in collection |

|  |
| --- |
| max(collection) |
| Returns largest element in collection |

|  |
| --- |
| min(collection) |
| Returns smallest element in collection |

|  |
| --- |
| any(collection) |
| returns True if any element in x corresponds to the Boolean True (see Boolean Values for correspondences) |

|  |
| --- |
| all(collection) |
| returns True if all elements in x corresponds to the Boolean True (see Boolean Values for correspondences) |

#### Iterators Based on Ordered Collections

|  |
| --- |
| enumerate(collection) |
| Returns iterator containing elements of collection preceded by their indexes |

|  |
| --- |
| zip(collection1, collection2): |
| Returns iterator containing elements from collection1 and collection2 at same index |

|  |  |
| --- | --- |
| reversed(collection) | |
| Returns iterator of items in collection in reverse order | |
| **Note** | Must wrap in list() to get the list itself |

|  |  |  |  |
| --- | --- | --- | --- |
| sorted(iterable, key=key, reverse=reverse) | | | |
| Returns a list of the items in iterable sorted in order | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| iterable | The iterable being sorted | No |  |
| key | An argument to determine the comparison method | Yes | None |
| reverse | A Boolean for if the larger values go first | Yes | False |
| **Note** | See Basic Sorting Functions for more info | | |

#### List Manipulation

|  |
| --- |
| list.append(argument) |
| Adds argument to the end of list |

|  |
| --- |
| list.clear() |
| Removes all items from list |

|  |
| --- |
| list.extend(iterable) |
| Adds all the items in iterable to the end of list |

|  |
| --- |
| list.insert(index, argument) |
| Adds argument to list at index |

|  |  |  |  |
| --- | --- | --- | --- |
| list.pop(index) | | | |
| Removes item at index in list and returns removed value | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| index | The list index of the item to be popped | Yes | The last item in list |

|  |
| --- |
| list.remove(argument) |
| Removes argument from list |

|  |
| --- |
| list.reverse() |
| Reverses the elements of list |

|  |  |  |  |
| --- | --- | --- | --- |
| list.sort(key=key, reverse=reverse) | | | |
| Returns list with the elements compared with less than (<) operators | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| key | An argument to determine the comparison method | Yes | None |
| reverse | A Boolean for if the larger values go first | Yes | False |
| **Note** | See Basic Sorting Functions for more info | | |

|  |
| --- |
| del list[index] |
| Removes item at index from list |

|  |
| --- |
| random.shuffle(list) |
| Rearranges the order of the elements in list |

#### Queues and Deques

|  |
| --- |
| deque.append(argument) |
| Adds argument to the end of object deque |

|  |
| --- |
| deque.popleft() |
| Removes element at the beginning of deque and removes returned value |

### Dictionary Manipulation

|  |
| --- |
| dictionary.get(key) |
| If key is a key is dictionary, ???; otherwise, returns None |

|  |
| --- |
| len(dictionary) |
| Returns the number of elements in dictionary |

|  |
| --- |
| list(dictionary) |
| Returns a list of keys in dictionary |

#### Dictionary Views

|  |
| --- |
| dictionary.items() |
| Returns an iterator of dictionary’s keys and values as tuples, see Dictionary Views for more info |

|  |
| --- |
| dictionary.keys() |
| Returns an iterator of dictionary’s keys, see Dictionary Views for more info |

|  |
| --- |
| dictionary.values() |
| Returns an iterator of dictionary’s values, see Dictionary Views for more info |

### Classes and OOP

|  |
| --- |
| type(argument) |
| Returns the data type of argument |

|  |
| --- |
| isinstance(argument, type) |
| Returns Boolean stating if argument is of type type |

|  |
| --- |
| issubclass(type 1, type 2) |
| Returns Boolean indicating if type 1 is a subclass of type 2 |

### Modules and Packages

|  |
| --- |
| dir(module) |
| Returns a list of all functions in module |

|  |
| --- |
| help(module) |
| Returns a manual page created from the docstrings in module |

|  |  |  |  |
| --- | --- | --- | --- |
| importlib.reload(argument) | | | |
| Reloads the previously loaded module or package importlib | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| argument | A module in importlib | Yes | Reloads all of importlib |

|  |
| --- |
| dir(argument) |
| Returns a list of the objects in argument |

|  |
| --- |
| dir() |
| Returns a list of the objects currently defined in the program |

### Control Flow Statements

|  |  |  |  |
| --- | --- | --- | --- |
| range(stop, start, step) | | | |
| Returns a tuple of integers beginning at start and including every step number after until stop-1 | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| stop | The first number not to be included in the sequence | No |  |
| start | The first number in the sequence | Yes | 0 |
| step | The counting interval | Yes | 1 |

### File I/O

|  |  |  |  |
| --- | --- | --- | --- |
| open(‘file’, ‘mode’) | | | |
| Opens the file file for interaction according to mode | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| file | The file path for the item to be opened; just the file name if the file and program are in the same folder |  |  |
| mode | The way to open the file—see Opening Files for options | Yes | r |
| **Note** | For more info, see Opening Files | | |

|  |
| --- |
| fileIO.close() |
| Closes the file opened in the variable fileIO |

|  |
| --- |
| os.getenv(argument) |
| Returns the path of the variable argument |

|  |
| --- |
| os.getcwd() |
| Returns the path to the current working directory |

#### Reading Files

|  |  |  |  |
| --- | --- | --- | --- |
| fileIO.read(size) | | | |
| Reads the contents of the file indicated by fileIO | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| size | The amount of the file to be read at a time | Yes | Complete file |
| **Note** | For more info, see Reading from Files | | |

|  |
| --- |
| fileIO.readlines() |
| Reads the contents of the file indicated by fileIO up through the next newline character |

##### Reading JSON

|  |  |  |  |
| --- | --- | --- | --- |
| json.load(fileIO, parse\_float=parse\_float, parse\_int=parse\_int) | | | |
| Reads a JSON file indicated by fileIO and converts it to Python data types | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| fileIO | The opened JSON file | No |  |
| parse\_float | The Python data type of JSON float objects | Yes | Float |
| parse\_int | The Python data type of JSON int objects | Yes | Integer |
| **Note** | See Reading JSON Files for more info | | |

|  |  |  |  |
| --- | --- | --- | --- |
| json.loads(string, parse\_float=parse\_float, parse\_int=parse\_int) | | | |
| Reads string which is formatted as a JSON and coverts it to Python data types | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| fileIO | The opened JSON file | No |  |
| parse\_float | The Python data type of decimals in string format/JSON float objects | Yes | Float |
| parse\_int | The Python data type of integers in string format/JSON int objects | Yes | Integer |
| **Note** | See Reading JSON Files for more info | | |

#### Writing Files

|  |
| --- |
| fileIO.writelines(line) |
| Writes line into the file opened in the variable fileIO |

|  |
| --- |
| fileIO.tell() |
| Returns the fileIO object’s position in the file it goes to as an opaque number (???) for a text file or a number of bytes from the beginning for a binary file |

|  |  |  |  |
| --- | --- | --- | --- |
| fileIO.seek(change, start) | | | |
| Changes the fileIO object’s position in the file it goes to by change bytes beginning at start | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| change | The number of bytes to change position; a negative number will move the position forward | No |  |
| start | The position the counting should start from | Yes | 0 |
| **Note** | The default for start is the beginning of the file; the other options are 1 (the current position) and 2 (the end of the file) and they can only be used with binary files. | | |

##### Writing JSON

|  |  |  |  |
| --- | --- | --- | --- |
| json.dump(object, fileIO, indent=indent) | | | |
| Writes object to the file indicated by fileIO as a JSON | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| object | The object to become a JSON | No |  |
| fileIO | Indicates in what file the JSON will be written | No |  |
| indent | The indentation level used for printing—see Writing JSON Files for values and their meanings | Yes | None |
| **Note** | See Writing JSON Files for more info; keyword arguments unlikely to be used not included here | | |

|  |  |  |  |
| --- | --- | --- | --- |
| json.dumps(object, indent=indent) | | | |
| Writes object to a string formatted as a JSON | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| object | The object to become a JSON-formatted string | No |  |
| indent | The indentation level used for printing—see Writing JSON Files for values and their meanings | Yes | None |
| **Note** | See Writing JSON Files for more info; keyword arguments unlikely to be used not included here | | |

### Web I/O

### Database I/O

### GUI Creation

|  |
| --- |
| Tk() |
| Create a tkinter GUI window |

#### Tkinter Widgets

|  |  |  |  |
| --- | --- | --- | --- |
| ttk.Button(parent, text=”text”, | | | |
| Create a button widget in parent | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| text | The text appearing in the button | Yes | None |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| ttk. | | | |
|  | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| parent | The parent widget | No |  |
| **Note** |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| widget.config(\*\*kwargs) | | | |
| Update the attributes of widget given in \*\*kwargs | | | |
| **Argument** | **Definition** | **Optional** | **Default** |
| widget | The widget being changed | No |  |
| \*\*kwargs | A keyword argument list of the attributes being changed and their new values | No |  |
| **Note** | See Widgets for more info | | |

### Errors

|  |  |
| --- | --- |
| sys.exc\_info() | |
| Returns info about an error | |
| **Note** | Primarily used in except statements of Error Handling with Try |

### Other

|  |
| --- |
| id(argument) |
| Returns the memory address of argument as an integer |

|  |
| --- |
| random.randint(x, y) |
| Returns a random integer between x and y |

|  |
| --- |
| del variable |
| Prompts garbage collection of variable |

|  |
| --- |
| sys.exit() |
| Terminates a script |

## Appendix D: Syntax Highlighting

Info on syntax and syntax highlighting

[https://stackoverflow.com/questions/54812670/vscode-configure-syntax-highlighting-to-match-a-style-guide?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/54812670/vscode-configure-syntax-highlighting-to-match-a-style-guide?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZO96sh8c$)

[https://stackoverflow.com/questions/40175795/how-to-add-keywords-syntax-highlighting-to-language-extension?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/40175795/how-to-add-keywords-syntax-highlighting-to-language-extension?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZJCNOXrI$)

[https://stackoverflow.com/questions/51534758/visual-studio-code-sql-syntax-highlighting-in-py-files?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/51534758/visual-studio-code-sql-syntax-highlighting-in-py-files?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZJd8PQjY$)

[https://stackoverflow.com/questions/55383756/adding-syntax-highlighting-from-different-extensions-in-vscode?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/55383756/adding-syntax-highlighting-from-different-extensions-in-vscode?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZD1n7Wb8$)

[https://stackoverflow.com/questions/58905614/lost-python-syntax-highlight-in-vscode?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/58905614/lost-python-syntax-highlight-in-vscode?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZtg9AniM$)

[https://stackoverflow.com/questions/56107128/dynamic-syntax-highlighting-in-vscode?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/56107128/dynamic-syntax-highlighting-in-vscode?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZjTT-9TM$)

[https://stackoverflow.com/questions/38148857/customizing-syntax-highlighting-in-visual-studio-code?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/38148857/customizing-syntax-highlighting-in-visual-studio-code?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZ2XbFIOo$)

[https://stackoverflow.com/questions/46904894/extending-vs-code-syntax-highlighting?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/46904894/extending-vs-code-syntax-highlighting?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZMukH79w$)

[https://stackoverflow.com/questions/59925791/vscode-language-support-syntax-highlighting?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/59925791/vscode-language-support-syntax-highlighting?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZBdo8aBM$)

[https://stackoverflow.com/questions/30775551/how-to-manually-set-language-for-syntax-highlighting-in-visual-studio-code?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/30775551/how-to-manually-set-language-for-syntax-highlighting-in-visual-studio-code?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZXgZ_eX4$)

[https://stackoverflow.com/questions/51605113/visual-studio-code-meaning-of-syntax-highlight-colors?r=SearchResults](https://urldefense.com/v3/__https:/stackoverflow.com/questions/51605113/visual-studio-code-meaning-of-syntax-highlight-colors?r=SearchResults__;!!PhOWcWs!jdnZqjKLIky08KVMUY0jomnJJ1NOKsjc3FoyIt3KC4RYDXCr_eY5AGqZp-prUyc$)

## Documentation Links

<https://docs.python.org/3.8/>

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

<https://www.python.org/dev/peps/pep-0257>

<https://pypi.org/>

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

<https://requests.readthedocs.io/>

<https://lxml.de/index.html#documentation>

<https://www.selenium.dev/selenium/docs/api/py/api.html>

<https://urllib3.readthedocs.io/>

<https://docs.scipy.org/doc/numpy/>

<https://pandas.pydata.org/pandas-docs/stable/>

<https://openpyxl.readthedocs.io/en/stable/>

<https://docs.djangoproject.com/>

1. “A scope is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.” [↑](#footnote-ref-1)