**Python Library**

**Types of Python Apps**

* Console App
  + Runs in the command prompt for an OS
* GUI App
  + Uses a GUI
* Web App
  + Runs in a browser

**Python Logistics**

* .py files are made in a code editor
* Source code is interpreted by a Python interpreter
* This code is compiled into bytecode
* The Python virtual machine translates the bytecode so it can be run by the OS
* The PVM is part of the Python interpreter, so it pretty much works on any platform

**Planning Apps/Code**

* Hierarchy charts # plan the modules of a program
  + Main function at top
  + The functions the main needs to call
  + Any functions those functions need to call
  + Remember that functions should be verbs, so module names should also be verbs (or sometimes objects of prepositions i.e. a module called convert\_temp uses to\_celsius …)
* Pseudocode # hand write out the code, using words rather than exact syntax

**Basic Syntax**

* No “;” at the end of statements like JS/PHP
* print() requires “” around anything you want to display
* Variable names are case sensitive
* In IDLE, F5 will compile and run a Python program
* The IDLE shell will run just like the command prompt would for testing an app
* Syntax and runtime errors are similar to other languages
* Coding over multiple lines
  + best practice to indent statements that cover multiple lines
  + Implicit indentation
    - break a line before or after operators like +/-
  + Explicit indentation
    - break a line by adding a backslash at the end
    - not encouraged or best practice
* *Shebang* line
  + 1st line of the program
  + Ignored by Windows, but Unix-like systems including OS X use it to determine what interpreter to use to run the program
  + #!/usr/bin/env python3 # shebang line that specifies python3
  + Good practice to always code this
* Comments
  + Anything appearing after a hash # is now a comment
  + Can code big blocks of comments with “””three double quotes at the start and at the end”””
    - These are called *docstrings* and are usually used as documentation for functions/modules
* Chaining functions
  + Code a function as the argument of another function:
    - miles = round(float(input(“Enter number of miles: “)), 2) # gets input, converts to float, rounds to 2 digits (NOT DECIMAL PLACES!)
* Calling methods
  + *variableName.methodName*()
  + Different than functions that accept args # str(*variableName*)

**Format for Modules/Applications**

* 1st line is shebang
* import any modules used by an app
* define any global constants if necessary
* store all other app code in functions
* define the main function last
* end using if statement that calls the main function if the module is the main module

**Data Types**

* str # string
* int # integer
* float #float (decimal)

**Variables**

* Names
  + Must begin with letter or underscore
  + No spaces and underscore is the only special char allowed
  + Can’t be python keywords
  + Use either camelCase or underscore\_notation for variable names
* Assign variables simply by using =
  + first\_name = “Chuck”
  + It is possible to overwrite variables defined in this way
* Are case sensitive
* If two variables are used, then second defines the first
  + num1 = num2 # results in num1 now having the same value as num2
* Scope
  + Use local scope whenever possible
  + can use global keyword if necessary to modify a global variable (NOT BEST PRATICE)
  + Ok to use global constants (though they can be modified so not really a constant) if you use ALL\_CAPS for the variable name

**Numbers**

* Mathematic operators
  + +, -, /, \*
  + // # integer division that drops/truncates the decimal
  + % # modulus that returns the remainder of division
  + \*\* # exponent that raises the left number to the power of the right
  + += # adds the result of the expression to the variable
  + -= # subtracts the result from the variable
  + \*= # multiplies the result to the variable
  + /= # the following are also available
  + //=
  + %=
  + \*\*=
* Scientific notation
  + 2.232e+5 # 223,200
  + 2.232e-5 #0.0000232
* Float vs Int
  + Int are exact numbers while floats are approximate numbers # can cause errors with floats
  + Good practice to always round floats to avoid this issue # using round(*num* or *calculation*, *digits*)
    - Or use the Decimal Module, outlined below
  + Floats can hold up to 16 significant figures
* Formatting numbers (for display)
  + use the .format() method of a string
  + Syntax
    - “{:*format\_specification*}”.format(*data*)
      * *format\_specification* syntax
        + [*field\_width*][comma][.*decimal\_places*][*type\_code*]
    - Type Codes
      * d # integer (decimals can’t be specified)
      * f # float (decimals can be specified, appears that numbers are rounded)
      * % # percent (multiplies by 100 and adds “%”)
      * e # sci not (converts the number to scientific notation
    - Formatting into columns
      * can pass multiple format\_specifications to multiple values (see final example)
      * by default, numbers are right aligned and strings are left aligned
        + can override this using < or > to point the direction to align (see example)
  + Examples
    - num = 12345.6789

print(“{:.2f}”.format(num)) # 12345.68 float with 2 decimal places (fw & comma omitted)

print(“{:.4f}”.format(num)) # 12345.6789 float with 4 decimal places

print(“{:,.2f}”.format(num)) # 12,345.68 comma included this time

print(“{:15,.2f}”.format(num)) # 12,345.68 adds spaces to the left to make width 15 chars total

* + - num = 12345

print(“{:d}”.format(num)) # 12345

print(“{:,d}”.format(num)) # 12,345

* + - num = 0.12345

print(“{:.0%}”.format(num)) # 12%

print(“{:.1%}”.format(num)) # 12.3%

* + - num = 12345.6789

print(“{:.2e}”.format(num)) # 1.23e+04

print(“{:.4e}”.format(num)) # 1.2346e+04

* + - print(“{:15} {:>10} {:>5}”.format(“Description”, “Price”, “Qty”)) # right algin price/qty

print(“{:15} {:10.2f} {:5d}”.format(“Hammer, 9.99, 3))

print(“{:15} {:10.2f} {:5d}”.format(“Nails, 14.5, 10))

Console # each field has set width, price/qty are forced to right align

Description Price Qty

Hammer 9.99 3

Nails 14.50 10

* Functions with numbers
  + int(*data*)
    - converts the data to int and returns the int value (truncates?)
  + float(*data*)
    - converts the data to float and returns the float value
  + max(*val1[, val2]*)
    - returns the maximum number of the two values
  + round(*number*[, *digits*])
    - rounds the number to the specified number of digits or to an integer if digits is blank
    - rounds to a number of digits, not number of decimal places
* Math Module
  + import math [as m]
  + Functions
    - m.pow(*num, pow*)
      * raises *num* to the specified *pow* # works like *num\*\*pow*
    - m.sqrt(*num*)
      * returns the sqrt of the *num*
    - m.ceil(*num*)
      * rounds the float up to the nearest int
      * to return a decimal, you can multiply by a factor of 10, ceil, then divide by same factor
    - m.floor(*num*)
      * rounds the float down to the nearest int
      * to return a decimal, you can multiply by a factor of 10, floor, then divide by same factor
  + m.pi
    - the value of pi to 15 decimal places
* Locale Module # format numbers/currency to other country’s formats
  + import locale [as lc]
  + Functions
    - lc.setlocale(*category, locale*) # must be set before calling other locale functions
      * sets locale for category(s) and returns a string for the locale
      * *category* options
        + LC\_ALL # set for all categories
        + LC\_NUMERIC # only numbers
        + LC\_MONETARY # only monetary values
      * *locale* options
        + can vary between Windows/Mac, and long codes
        + can use if statements to check if setlocale() returns “C”
        + us # short code en\_US # long code
        + uk en\_UK
        + de # Germany de\_DE
      * if *locale* is an empty string, attempts to set to user’s default locale, if that isn’t possible, returns a code of “C”
      * Example:
        + result = lc.setlocate(lc.LC\_ALL, “”) # works on most Windows sys.

if result == “C”:

lc.setlocale(lc.LC\_ALL, “en\_US”) # sets locale if first version fails

* + - lc.currency(*num*[, *grouping*])
      * returns the specified *num* formateed as currency
      * if *grouping*=True is set, thousands separators are used
      * Example
        + print(lc.currency(12345.15, grouping=True)) # $12,345.15 (for US)
    - lc.format(*format, num*[, *grouping*])
      * *format* works similarly to *format\_specification* from .format() method of a string
      * Example
        + print(lc.format(“%d”, 12345, grouping=True)) # $12,345 (for US)
        + print(lc.format(“%.2f, 12345.15, grouping=True)) # 12,345.15 (for US)
* **Decimal Module**
  + Typically only used for financial calculations, and usually just round floats for other progs
    - Decimal objects use more memory and take more time to crunch
  + You need the decimal class from the decimal module
    - from decimal import Decimal # allows you to construct *Decimal objects*
  + Syntax for constructing a Decimal object
    - *var\_name* = Decimal(“*data*”) # you supply the number as a string in “”
  + Can then use those objects in standard calculations and mix with *int* data types, as both types represent exact values
  + Methods
    - *decimal\_object*.quantize(Decimal(“positions\_code”)[, rounding\_constant])
      * used to round numbers
      * rounds your *decimal\_object* to the number of decimal places specified by your “positions\_code”
        + positions\_code

1.0 # specifies 1 decimal place

1.00 # 2 decimal places

1.000 # 3 decimal places (works if 9.999 too, but 1.000 is better)

* + - * rounding constant determines how rounding is performed (when to go up or down if digit determining rounding is a 5)
        + need to import these

from decimal import *ROUND\_HALF\_...*

* + - * + **ROUND\_HALF\_UP # used for most business math (rounds 5 up)**
        + ROUND\_HALF\_DOWN # rounds 5 down
        + ROUND\_HALF\_EVEN # default (rounds 5 up for odd and down for even)

**Strings**

* Can be single or double quotes
  + “This apostrophe doesn’t need escaping”
  + ‘This apostrophe can\’t not be escaped’
  + ‘Type “x” to exit’ # is acceptable
* Emtpy string has nothing between the quotes
* Concatenate using + or +=
  + Can mix variables with “quoted text “ when concatenating
  + Must convert numbers to strings using str() if using a variable with numeric data
* Special characters needing to be escaped
  + \n # codes a new line
  + \t # codes a tab
  + \r # codes a return
  + \” # quotation mark in a double quoted string
  + \’ # quotation mark in a single quoted string
  + \\ # backslash
* Strings consist of *Unicode* chars
  + this maps each char to an integer or *ordinal* value
  + these values determine the sort order (why 0-9 before A-Z before a-z)
  + see *ord(char)* function for more on this
* Accessing a char within a string
  + chars are mapped just like list items # cannot alter a char though (see example)
    - Example
      * message = “Hello world!”
      * message[0] # “H”
      * message[-1] # “!”
      * message[16] # IndexError: string index out of range
      * message[0] = “J” # TypeError: string is immutable
* Slicing a string # works like with lists
  + *string*[*start:end:step*]
  + returns chars beginning with *start*, then returns every char before *end*, using each *step* (default is 1)
    - Example
      * message = “Hello out there!”
      * message[:5] # “Hello” returns indexes 0-4
      * message[6:9] # “out” returns 6-8
      * message[10:] # “there!”
      * message[:-1] # “Hello out there” returns everything but stops before the end
* Can multiply chars just like with strings
  + Example
    - print(“=” \* 20) # ====================
    - print(“A horse! ” \* 2) # A horse! A horse!
* Searching strings for terms
  + *term* in *string* # searches the *string* for the supplied *term* (case sensitive)
  + often used with if statement # if *term* in *string*: … and can use variables for term
* Looping through strings
  + works like other loops
  + Example
    - for char in *string*: # use *char* from this example in the loop statements

*statements…*

* Find and replace parts of strings
  + .find(*str*[, *start*][, *end*]
    - searches for the specified substring and returns the index of the first occurrence or -1 if the string isn’t found, with optional *start* and *end* parameters (indexes)
  + .repace(*old, new*[, *num*])
    - returns a new string with occurrences of the old substring replaced by the new substring
    - optional third parameter specifies the number of occurrences to replace (starts with beginning and counts how many to replace)
* Functions with strings
  + str(*input*) # converts other data types (your *input*)to strings
  + ord(“*char*”) # returns the integer (ordinal) value for the Unicode *char* supplied
  + len(*str*) # returns the length for the specified *str*
* Methods of strings
  + .isalpha(*str*) # returns True if all chars are alphabetic
  + .islower(*str*) # returns True if all chars are lowercase
  + .isupper(*str*) # returns True if all chars are uppercase
  + .isdigit(*str*) # returns True if all chars are digits 0-9
    - The four above methods have (*str*), but are used *str.method()* in examples
  + .startswith(*str*) # returns True if the primary string starts with the specified *str*
  + .endswith(*str*) # returns True if the primary string ends with the specified *str*
  + .lower() # converts all chars to lowercase
  + .upper() # converts all chars to uppercase
  + .title() # converts the string to title case (all letters capital including prepositions)
  + .lstrip() # strips whitespace from the left
  + .rstrip() # strips whitespace form the right
  + .strip() # strips whitespace from both sides
  + .ljust(*width*) # returns a left-justified string with spaces added to fill the width
  + .rjust(*width*) # returns a right-justified string with spaces added to fill the width
  + .center(*width*) # returns a centered string with spaces added to fill the width
  + .find(*str*[, *start*][, *end*]
    - searches for the specified substring and returns the index of the first occurrence or -1 if the string isn’t found, with optional *start* and *end* parameters (indexes)
  + .repace(*old, new*[, *num*])
    - returns a new string with occurrences of the old substring replaced by the new substring
    - optional third parameter specifies the number of occurrences to replace (starts with beginning and counts how many to replace)
  + .split([*delimiter*][, *num*])
    - splits a string into substrings at each *delimiter* and returns a list of those substings
    - default delimiter is any whitespace
    - optional *num* parameter to specify the number of occurrences to replace
  + .join(*sequence*)
    - Remember that concatenating strings is done with + or +=, and this is different
    - joins the elements of a *sequence* into a string that uses the current string as the delimiter
    - works well to join list items into a string
      * Example
        + address = [“John Doe”, “1234 Any Street”, “New York”, “NY”, “10001”]
        + address = “|”.join(address)
        + print(address) # John Doe|1234 Any Street|New York|NY|10001
    - Also works on strings
      * Example
        + letters = “HORSE”
        + letters\_spaced = “ “.join(letters)
        + print(letters\_spaced) # H O R S E

**Lists**

* “Lists” are essentially arrays with “items” instead of elements
* Python uses 0 indexing
* Lists are mutable
  + They do not have to be returned when modified by a function (they simply get modified)
  + If a list stored in a variable is assigned to a new variable, both variables refer to the same list
    - modifying one will modify the list to which both variables refer (it’s one list, stored in two places, and either place can modify it)
  + Use the deepcopy(*list*) method of the copy module to produce a new list
    - import copy

list\_two = copy.deepcopy(list\_one) # this will result in two different lists

* Creating lists
  + *list\_name* = [] # creates an empty list
  + *list\_name* = [*item1, item2…*] # creates a list and adds items at the same time
  + *list\_name* = [“e”] \* 5 # list\_name = [“e”, “e”, “e”, “e”, “e”]
  + Contrast vs tuple creation which uses () instead of []
    - tuples are immutable and lists are mutable
* Lists can include any type of data
* Can modify an entry in a list by setting a new value
  + *list\_name*[3] = 18.8 # the fourth entry of the list is now 18.8
* Accessing entries
  + *var\_name* = *list\_name*[3] # var\_name now has a value of 18.8 (for example)
* Processing lists
  + While loops
    - while *i* < len(*list\_name*):

code to execute using *list\_name*[i] # must use i as the index value then increment

i += 1 # increment counter

* + For loops
    - for *i* in *list\_name*:

code to execute simply using *i* to represent each entry

# often easier than the while loop example, because no index or list length needed

* + Nested statements can be used to process multidimensional lists
    - for *list* in *lists*:

for *item* in *list*:

print(item)

* Multidimensional lists
  + Can create a “list of lists” # like a two dimensional array
  + Accessing items is just like with JS or PHP
    - *list\_name*[*first\_index*][*second\_index*] # etc to exend to more dimensions
* Created functions that accept list objects
  + because lists are mutable (str, int, float, and bool are immutable) you do not have to return a list
  + if you def a function that accepts a list argument and you append, pop, etc. the list is modified without having to return it
* Slicing a list
  + *list\_name*[*start:end:step*]
    - returns the list beginning with *start* index
      * starts at index 0 if start is omitted # *list\_name*[:*end:step*]
    - last value returned is the item ***before*** the *end* index
      * returns to the end of the list if end is omitted #*list\_name*[4:]
    - *step* can be used to reverse the order or skip entries
      * *step* of 2 will return items 0, 2, 4…
      * *step* of -1 will return all items in reverse order
  + Can leave multiple things blank #*list\_name*[::-1] *list\_name*[4::2]
* Concatenate lists
  + *new\_list = list1 + list2* # creates a new list that appends list2 items to list1
  + *list1 += list2* # appeds list2 items to list1, modifying list1 in the process
* Functions to use with lists
  + len(*list*)
    - returns the number of item in the list
  + min(*list*)
    - returns the min value in the list
  + max(*list*)
    - returns the max value in the list
  + sorted(*list*[, *key=function*])
    - returns a new list consisting of the sorted items of the original list, with optional key function to be called on items before sorting
      * key=str.lower # levels playing field of upper/lower cases but does not actually change any of the list cases to lower case
    - key difference between this and .sort is that sorted returns a new list and .sort modifies a list
      * must store your new list in a variable
  + random.choice(*list*) # from the random module
    - returns a randomly selected item from the list
  + random.shuffle(*list*) # from the random module
    - shuffles the items in the list randomly
  + copy.deepcopy(*list*)
    - in the copy module
    - will produce a copy of a list (not just another reference to it)
  + can print(*list*) the list to the console
  + can print each item on a new line
    - for *item* in *list\_name*:

print(*item*)

* Methods of a List # *list\_name.method\_name*([*args*])
  + .append(*item*)
    - appends the *item* to the end of the list
  + .insert(*index, item*)
    - inserts the *item* at the specified *index* and shifts everything else down the list
  + .remove(*item*)
    - removes the 1st item in the list that is equal to the specified item and shifts everything up
  + .index(*item*)
    - returns the index of the first occurrence of the item
  + .pop([*index*])
    - removes the item at the specified index and return it (can store in a variable), if none specified, removes the last item from the list
    - can use in conjunction with .index to find, remove, and return a specific entry
  + .count(*item*)
    - returns the number of occurrences of an item in the list (or 0 if it’s not found)
  + .reverse(*list*)
    - returns and reverses the order of the items in a list
  + .sort([*key=function*])
    - sorts the list items in place and modifies your list
      * use sorted function above to return a new list without modifying your list
    - optional key arg specifies a function to be called on each item before sorting
      * using key=str.lower will level case playing field without actually changing case of entries
    - remember that sort order is 0-9, A-Z, a-z

**Tuples**

* Tuples are similar to lists but are immutable and can’t be altered
* Creating a tuple # like a list, but uses () instead of []
  + *tuple\_name* = (*item1, item2…*)
  + To create a tuple with a single item you must have a comma
    - *tuple\_name =* (*item1,*) # not sure why you might do this
    - omit the comma and it is simply an int
* Access a tuple
  + You access tuples the same as lists, using []
  + You cannot modify items, however
* Unpacking a tuple
  + a, b, c = *tuple\_name* # must have same number of variables as the tuple length
* Returning a tuple
  + if you end a function with return x, y, z then your returned object is a tuple
  + def return\_tuple()

# code that determines the value of x, y, z

return x, y, z

new\_tuple = return\_tuple()

x, y, z = return\_tuple() # unpacks it all at once

**Dictionaries**

* A dictionary is basically just an associative array
* Lists are *ordered* or indexed, while dictionaries are *unordered*
  + cannot loop through a dictionary and expect it to be in order, even when using numbers as keys
* Syntax for creating a dictionary
  + *dictionary\_name* = {*key1:value1, key2:value2* …}
    - whitespace is ignored, so can put key:value pairs on multiple lines for readability
  + can create an empty dictionary just using empty {}
    - keys
      * can be numbers or strings # strings in “” and numbers not in quotes
      * immutable
    - values
      * strings go in “”
      * numbers do not
      * can be lists or other dictionaries
* Accessing dictionary items
  + same as with lists or associative arrays
    - *dictionary\_name*[*key*] # string keys must be in “”
  + if the key doesn’t exist, a KeyError will occur
  + can set the values of a dictionary by assigning a value
    - *dictionary\_name*[*key*] = *value*
    - will add new entries this way or change values
* Checking or searching a dictionary for a key
  + *key* in *dictionary\_name*
    - if *key* in *dictionary\_name*: # for example
* Deleting dictionary items
  + del *dictionary\_name*[*key*]
    - throws a KeyError if the key doesn’t exist
  + .pop(*key*[, *default\_value*])
    - returns the value of the specified key and deletes the key/value pair
    - throws a KeyError if the key doesn’t exist, unless you specify a *default\_value* to return otherwise
  + .clear()
    - deletes all dictionary items
    - doing *dictionary\_name* = {} will cause the variable to refer to a new dictionary, and does not delete any items
* Methods of dictionaries
  + .get(*key*[, *default\_value*])
    - returns the value of the *key*, or returns None/*default\_value* if the key doesn’t exist
  + .pop(*key*[, *default\_value*])
    - returns the value of the specified key and deletes the key/value pair
    - throws a KeyError if the key doesn’t exist, unless you specify a *default\_value* to return otherwise
  + .clear()
    - deletes all dictionary items
    - doing *dictionary\_name* = {} will cause the variable to refer to a new dictionary, and does not delete any items
  + .keys()
    - returns a view object that contins all of the keys in the dictionary (default iterator for a dictionary)
  + .items()
    - returns a view object that contains a tuple for each key/value pair in the dictionary
  + .values()
    - returns a view object that contains all of the values in the dictionary
* **Looping through dictionaries**
  + Need to use an iterator (view objects created by the .keys(), .items(), or .values() methods)
  + The default iterator is .keys(), so when using for loops, the *var* after “for” is used as the key
  + If needing to sort items in a dictionary, see section below on creating a list for the keys, then looping
  + Examples
    - for *i* in *dictionary\_name*: # same result as using *dictionary\_name.*keys()
      * *statements* using *dictionary\_name*[*i*] to loop
    - **for *i, y* in *dictionary\_name*.items():**
      * ***statements* using *i* and *y***
      * ***i* represents the key, while *y* represents each value**
    - for *i* in *dictionary\_name.*values():
      * *statements* using *i* # only works with values, unable to access contents of the keys
* Converting between dictionaries and lists
  + Necessary to convert a dictionary to a list to sort its contents
  + Constructors
    - list(*view*)
      * converts the view object into a list
    - dict(*list*)
      * converts the specified two-dimensional list or tuple to a dictionary
      * each row must contain exactly two values
        + *list* = [[*key1, value1*], [*key2, value2*] …]
  + Can store the keys from a dictionary in a list, sort that list, then use those sorted keys to display dictionary values
  + Example
    - *sorted\_keys* = list(*dicionary\_name*.keys())
    - *sorted\_keys*.sort()
    - for *sorted\_key* in *sorted\_keys*:
      * print(*sorted\_key* + “ “ + *dictionary\_name*[*sorted\_key*])
    - this will use the keys in a sorted order along with the original dictionary
* Complex dictionaries # multi-dimensional
  + Can store lists, tuples, or other dictionaries as values # still need a key for each entry
  + Accessing values
    - dictionary with dictionaries as values
      * *dictionary\_name*[*key1\_first\_level*][*key1\_second\_level*]
        + returns the first value (*value1\_second\_level*) from the first dictionary entry (*value1\_first\_level*)
        + can still produce a KeyError if a key doesn’t exist, use the get() method to return None rather than producing a KeyError
      * *dictionary\_name*.get(*key1\_first\_level*).get(*key1\_second\_level*)
        + returns the first value (*value1\_second\_level*) from the first dictionary entry (*value1\_first\_level*) or None if the second key doesn’t exist
        + to avoid problems with the first key not existing, you can code an empty dictionary as the second parameter of the first get() method

*dictionary\_name*.get(*key1\_first\_level*, {}).get(*key1\_second\_level*)

* + - dictionary with lists as values
      * *dictionary\_name*[*key1\_first\_level*][*list\_index*]
        + provide the index for the list to access the value
      * should check if *key1\_first\_level* exists first to avoid KeyError
  + Checking if a key exists in a 2nd level
    - key = “*second\_level\_key\_name*”
    - if key in *dictionary\_name*[“*first\_level\_key\_name*”]: # followed by statements to execute

**Dates and Times**

* The datetime module
  + import the date, time, and datetime classes
    - from datetime import date
    - from datetime import time
    - from datetime import datetime
    - from datetime import timedelta # changes or spans of time
  + Aware vs Naïve dates/times
    - naïve means objects are not *aware* of times zones or daylight savings
    - set optional *tzinfo* parameter of the datetime() and time() constructors to be *aware*
    - check documentation of the datetime module for more details
  + Constructors for date/time objects # other methods to create these objects are found below
    - date(*year, month, day*)
      * returns a date object to the specified date
      * date.today() # returns today’s date
    - time([*hour*][, *min*][, *sec*][, *microsec*])
      * returns a time object set to specified time
      * all time args are optional and default to 0
      * recall that 1,000,000 microsec / sec or 1 s = 1 x 106 µs
    - datetime(*year, month, day*[, *hour*][, *min*][, *sec*][, *microsec*])
      * returns a datetime object for the specified date/time
      * all time args are optional and default to 0
      * datetime.now() # returns current date and time
    - timedelta([*days*][, *seconds*][, *microseconds*][, *milliseconds*][, *minutes*][, *hours*][, *weeks*])
      * Syntax
        + three\_weeks = timedelta(*weeks=*3) # use named args

when specifying the length of time like this, order is not important

omit any units not using

* + Adding/Subtracting Time
    - combine date/time objects with timedelta objects
      * can add or subtract spans of time in this way
        + Example for three weeks from today

twft = date.today() + timedelta(weeks=3)

* + - subtract to get the time in between two dates/times
      * creates a timedelta object
        + Example

halloween = datetime(2019, 10, 31)

time\_span = halloween - datetime.now()

# time\_span is a timedelta object

# time\_span is negative if current date is after 10/31/2019

* + Attributes of date/time and timedelta objects
    - Syntax
      * *object\_name*.*attribute* # will return the value
    - timedelta
      * .days # returns number of days
      * .seconds # returns number of seconds in addition to days (remainder of seconds)
      * .microseconds # number of microseconds in addition to days and seconds (remainder of µs)
      * use the total\_seconds() method (see below) to access total # sec + µs
    - date/time object attributes
      * .year # returns 4-digit year
      * .month # returns the month as digit 1-12
      * .day # returns the day as digit 1-31
      * .hour # returns the hour as number 0-23
      * .minute # returns the minutes as number 0-59
      * .second # returns the second as number 0-59
      * .microsecond # returns the microsecond as number 0-9999999
  + Comparing date/time objects
    - greater than “>” means after # if datetime.now() > dt\_object that means it has passed
    - less than “<” means before
  + Methods
    - date objects
      * date.today() # current date (date object w/ no time)
    - datetime objects
      * datetime.now() # current date and time
      * datetime.strptime(*datetime\_str, format\_str*) # string parse time
        + uses the specified format string to convert the date/time string to a datetime object (creates a datetime object)
        + uses the common format string codes listed below
        + Example

halloween = datetime.strptime(“1988-31-10”, “%Y, %d, %m”)

the *format\_str* is used to interpret the *datetime\_str*

* + - timedelta objects
      * total\_seconds()
        + the total number of seconds and microseconds (for the entire length of time)

as a single floating point number

* + - * see attributes section for accessing timedelta attributes of this object
    - All date/time objects
      * strftime(*format\_str*)
        + converts a date/time object to a formatted string using *format\_str*
        + used to display date/time or store as a string
        + uses format string codes
  + Format String Codes
    - %d # day of month as a two digit number
    - %m # month as a two digit number
    - %y # 2-digit year
    - %Y # 4-digit year
    - %H # hour of the day in 24 hour format
    - %I # hour of the day in 12 hour format (don’t use to create times/datetimes)
      * %p # AM/PM specifier (AM)
    - %f # microsecond
    - %M # minite as number
    - %S # second as number
    - %a # abbr weekday name (Sat)
    - %A # full weekday name
    - %b # abbr month name (Oct)
    - %B # full month name
    - %c # date and time formatted for locale
    - %x # date formatted for locale
    - %X # time formatted for locale

**Built-In Functions**

* print([*data*[, *sep*[, *end*]]])
  + Displays any supplied *data* # if no *data* is passed, prints a blank line
    - *sep*=’’ # optional and will separate multiple arguments by char(s) within the quotes
    - *end*=’’ # optional and will append char(s) in quotes to the end of the statement
      * default *end* is \n
  + One or more arguments are printed directly to the console
    - Separating arguments with a comma will result in output without commas but with a space between each argument
    - print(1, 2, 3, 4) # 1 2 3 4
    - print(“Price:”, 19.99) # Price: 19.99
  + Can print() one argument as a concatenation
    - print(“Price: “ + “19.99”) # Price: 19.99
    - Don’t forget to use str() for numeric variables
  + Useful to add \n to print information on a new line
    - # usually \njust preceding your first word on new line
* input([*prompt*])
  + Pauses the program and waits for the user to input data
  + Data entered by the user is returned as a string when “enter” is pressed even if input is numeric
  + The optional *prompt* is printed to the console before pausing
  + Inputs must be processed, and are usually assigned to a variable (first\_name = input(“Enter your first name: ”))

**Conditional Statements**

* Relational operators
  + == # equal to
  + != # not equal to
  + > # greater than
  + < # less than
  + >= # greater than or equal to
  + <= # less than or equal to
* Logical operators (in order of precedence)
  + not
  + and
  + or
* Syntax
  + (age >= 65 and status == “retired”) or not age >= 18 # could have said age < 18, just illustrating here
* Strings
  + Sort order for > or <
    - digits lowest, then uppercase letters, then lowercase letters
      * so “Peach” is before “apple” which is weird
      * 0-9, A-Z, a-z
  + Use lower() or upper() to convert all letters to one version before comparing

**If Statements**

* General syntax
  + if *Boolean\_expression*:

*statements…*

[elif *Boolean\_expression*:

*statements…*]…

[else:

*statements…*]

* + **The statement line or code to run ends when the indentation ends for that block**
  + Multiple elif statements can be included
  + Can code a *pass* statement if the code should do nothing, should a Boolean evaluate true
    - pass # the only thing you type for the statement should be pass
  + Way to check if a value is in a list
    - if *value\_to\_check* in *list\_name*:

do so and so

if *value\_to\_check* not in *list\_name*:

do so and so

* Nested statements
  + Can nest if statements, and the nested tests are only performed if the outside condition evaluates true
  + Can sometimes simply code this way rather than using logical operators
* Example:
  + birth\_month = input("What month were you born?")

birth\_day = input("What day of the month were you born?")

if birth\_month == "March" and int(birth\_day) >= 21: # and statements here

print("Your sign is Aries")

if birth\_month == "April" and int(birth\_day) <= 19: # input needs converting

print("Your sign is Aries")

if birth\_month == "April": # nested if statements here

if int(birth\_day) >= 20:

print("Your sign is Taurus")

if birth\_month == "May":

if int(birth\_day) <= 20:

print("Your sign is Taurus")

if (birth\_month == "July" and int(birth\_day) >= 23) or (birth\_month == "August"

and int(birth\_day) <= 22):

print("Your sign is Leo") # Leo code split over two lines using or statements

**Loops**

* While loops
  + General syntax
    - while *Boolean\_expression*:

*statements…*

*statements to execute when Boolean is no longer true (or next line of code)*

* + Ending while loops
    - Can say “while True:” then use if statements within to specify what happens. to end the loop you should have one or more if statements that end in “return”. no value has to be returned, but the return statement will end the loop when that condition is met.
  + Example
    - i = 0 # initialize i
    - while *i* < len(*list\_name*):

code to execute for each list item using *list\_name*[i]

i += 1 # increment the counter

* For loops
  + General syntax
    - for *i* in *range*:

*statements…*

* + - *i* is a temporary variable that holds the value through each iteration
      * don’t have to reference *i* and if you don’t, it will simply run through the code *range* times
    - **range(*arg[s]*) is often used as the “range” above**
      * Can supply different combinations of *arg[s]* for different results
      * This could be an integer
        + for i in range(6): # i starts as 0 and the last i is 5
        + for i in range(6): means same as:

for (i=0; i < 6; i++)

* + - * can also specify a true range
        + for i in range(2, 6): means the same as

for (i=2; i < 6; i++)

* + - * can specify a step as well
        + for i in range(2, 6, 2): means the same as

for (i=2; i < 6; i+=2) # step now 2

* + - * can also supply a variable or len(*str* or *list*)
    - Lists
      * for *i* in *list:*

code to do with each list item # code automatically gets each item as i each time # so this works better than the while loop example above in some cases

* + - * for *temp\_variable* in *list*:

if …….elif…..break

* + - Enumerating a List # not for dictionaries
      * for *index, var* in enumerate(*list*): # usually used with an index for a list

print(“index “ + str(*index*) + “: ” + str(*var*))

* + - Dictionaries
      * for *key, value* in *dictionary*.items(): # need the .items() method to loop dictionaries

*statements* with *key* and *value*

* + - * make sure you remember that dictionaries are unordered, so order may change!
  + Can add “else:” statement at the end of a for loop for code to execute when finished
    - for *i* in *range*:
      * *statements*
    - else:
      * *statements*
* continue statements
  + continue # if typed at the end of a statement, will jump to top of loop & reevaluate condition
* break statements
  + break # at the end of a statement, will end the loop then execute the statement after the loop
  + common to add if statement with a break to end a loop when it’s met
* Ending an infinite loop using ctrl or cmd c

**Functions**

* Defining a function
  + Good practice to start function names with verbs
  + def *function\_name*([args]):

*statements*

* + Assign a default for an argument by assigning a value when defining a function
    - def *function\_name*(arg=7):

*statements*

* Calling a function
  + *function\_name*([args])
    - The *args* must be in the same order as when the function is defined
  + You can call a function and use “named arguments” by assigning a value to each argument
    - When you do this, the args do NOT have to be in the same order they were defined
      * *variable\_name = function\_name*(

*arg1=value1,*

*arg2=value2,*

*arg3=value3)*

* Need to include a *return* statement if a function should produce output
  + Usually assign the function to a variable when calling it (or print it) when producing output
* **The *main* *function***
  + It is good practice to put all code for a program into functions
  + Anything that isn’t part of a specific function goes into the *main function*
  + The main function starts the operation of the program
  + **Calling the main function**
    - if \_\_name\_\_ == “\_\_main\_\_”: # if this is the main module

main() # call the main() function

* + - The code above is important when reusing modules (sets of functions) in multiple programs
      * The main() function will not run when importing the module in a different program
      * Can use the main() function to test functions in the module rather than run a program

**Regular Expressions**

* Use the “re” module
  + import re
* Your regex always goes in “” # **Examples**
  + see examples below # used for the *exp* part with the functions below
  + your *exp* can be an exact match # “ain” searches for the letters ‘ain’ together
  + *exp* can include special sequences # “\bS\w+” looks for S at beginning followed by 1 or more chars
  + “^The.\*Spain$” looks for ‘The’ at beginning of str and ‘Spain’ at the end with 0 or more chars btw
  + This is similar to PHP, and can look there if needed for help
* For all examples
  + str = “The rain in Spain”
* Match Objects
  + contain information about the search/result
  + Methods
    - .span() # returns a tuple containing the start and end positions of the match
      * the first value is the start char, the second value is the char after the end of the word
      * Example
        + x = re.search(“\bS\w+”, str) # S at beginning followed by >= 1 chars
        + print(x.span()) # prints (12, 17)
    - .start() # returns the starting position
      * Example
        + x = re.search(“\s”, str) # print(x.start()) would return 3 (first whitespace)
    - .group() # returns the part of the string where there was a match
      * Example
        + using example from .span() above, print(x.group()) returns Spain
  + Attributes
    - .string # returns the string that was passed to the function ‘The rain in Spain’
* Functions
  + re.findall(*exp, str*)
    - returns a list (to be stored in a variable) that contains all matches
    - if the expression is found 3 times, your list has that expression 3 times
    - returns an empty list if no match is found
      * Example:
        + x = re.findall(“\Bain”, str) # x = [‘ain’, ‘ain’]
        + finds any matches of ‘ain’ not at the beginning \B
  + re.search(*exp, str*)
    - searches the str and returns a **match object** that contains information about the search/result
    - returns None if no match found
    - can use methods of the match object to access this information
  + re.split(*exp, str*[, *max*])
    - returns a list where the string has been split by each match
    - can supply optional *max* parameter for the max num of times to split
    - Example
      * x = re.split(“\s”, str) # \s is whitespace
      * print(x) # returns [‘The’, ‘rain’, ‘in’, ‘Spain’]
      * if re.split(“\s”, str, 1) # returns [‘The’, ‘rain in Spain’]
  + re.sub(*exp, sub, str*[, *max*])
    - returns a new string that replaces the *exp* with a *sub* at each location in the str
    - you can replace it only *max* times in the string if optional para is supplied
  + re.match()
* Chars
  + [] # specifies a set of chars # chars within here have some special meaning (like ^)
    - ^ means not if it’s the first char in the set, otherwise it means caret character
    - [arn] # returns a match where one of the chars (a, r, or n) is present
    - [^arn] # returns a match of any char except a, r, or n
    - [a-n] # returns a match where any lowercase letter from a to n
    - [^a-n] # returns a match of any char except a-n
    - [0-9][0-9] # match of 2 digit 00-99
    - [a-zA-Z] # any upper or lowercase letter
  + \ # signals a special sequence (or used as escape character)
  + . # any char except a newline char
  + ^ # str starts with # different than \b which can look at beginning of every word
  + $ # str ends with # different than \b which can look at the end of every word
  + \* # 0 or more occurrences
  + + # 1 or more occurrences
  + {} # exactly the specified number of occurrences {2} for 2 occurrences
    - a{m,n} # matches a repeating between m and n times
    - omit m and your lower bound is m=0 # a{,n}
    - omit n and your upper bound is infinite # a{m,}
    - can include a char after # a{m,}b will match m to infinity a’s followed by a b
  + | # either or
  + ? # matches 0 or more of the previous chars so ab? matches either a or b
    - also turns a statement into only matching as few things as possible
    - <.\*?> will match only first occurrence of anything inside <>
  + () # capture and group
* Special Sequences
  + \A # returns a match if the chars are at the beginning of a string
    - “\AThe”
  + \b # returns a match where the specified chars are at the beginning or the end of a word
    - ”\bain” # “ain” at the beginning
    - ”ain\b” # “ain” at the end
  + \B # returns a match where the chars are present, but NOT at the beginning or end of a word
    - ”\Bain” # “ain” present but not beginning
    - ”ain\B” # “ain” present but not the end
  + \d # a digit 0-9 # “\d”
  + \D # not a digit #”\D”
  + \s # whitespace char
  + \S # not a whitespace char
  + \w # any char a-z, A-Z, 0-9, or underscore (\_)
  + \W # not any of the chars for \w
  + \Z # returns a match if chars are at the end of a string “Spain\Z” (“Spain” the last word of the string)

**Recursive Functions**

* Recursion occurs when a function calls itself
  + can be used to execute the same action repeatedly until a certain condition is met
  + iterative functions (like while and for loops) only ever push onto the stack once, freeing up memory used when they complete, recursive functions keep pushing onto the stack and since they never end, the stack fills up until all memory is used and it stops, **this is why iterative functions are usually better**!
  + In IDLE, you can stop this with “CRTL+C” on Windows and “CMD+C” on Mac
  + Recursion depth, is the number of calls on the stack
    - Python’s max depth is 1000 # RecursionError if you exceed this
* Will run until it reaches the simplest form of the problem or the *base case*
  + each call that can’t be performed yet is a *deferred action*
  + once the *base case* is reached and Python can solve that problem, it completes the *deferred actions*
* Factorials
  + Recursive functions are commonly used for factorials
    - factorial is multiplying a number by all of the integers below it to 0
    - 0! = 1 # 0 is weird tho
    - 1! = 1 \* 1 = 1
    - 2! = 2 \* 1 = 2
    - 3! = 3 \* 2 \* 1 = 6 and so on
  + The recursive function
    - def factorial(num):
      * if num == 0:
        + return 1
      * else:
        + return num \* **factorial(num - 1)** # bold part is recursive
    - this will add calls on the stack until it gets to num = 0, then it completes all the calls and performs the multiplication
  + The iterative function
    - def factorial(num):
      * fact = 1
      * for number in range(1, num+1):
        + fact = number \* fact
      * return fact
* Fibonacci series
  + Used as an example of recursion in many textbooks
  + Rules of the sequence
    - the first number is 0
    - the second number is 1
    - every other number is the sum of the previous 2
  + Because of the rules, it helps to work backwards before performing any calculations
  + The recursive function
    - def fib(n):
      * if n == 0:
        + return 0
      * elif n == 1:
        + return 1
      * else:
        + return **fib(n-1) + fib(n -2)** # recursive part
    - very inefficient because it performs some calculations multiple times, but code is much simpler and easier to read
  + The iterative function
    - def fib(n):
      * if n == 0:
        + return 0
      * elif n == 1:
        + return 1
      * n1 = 0
      * n2 = 1
      * fib = 0
      * for i in range(2, n+1)
        + fib = n1 + n2
        + n1 = n2
        + n2 = fib
      * return fib
  + printing a set of numbers using these functions
    - for i in range(*how\_many\_nums\_to\_print*):
      * print(fib(i), end=”, “)

**Modules**

* There are additional modules for math operations in the Numbers section above
* See Format for Modules/Apps section at the beginning of this library for formatting of modules
* .py file that contains reusable code, usually in functions (like a JS library)
* Storage location for modules
  + Same directory as other modules that use them
  + Central location that’s in a *search path* (google how to set the search path)
* Documentation
  + Use docstrings (three double quotes) at the start/end of documentation (like block comments)
  + Indent docstrings for individual functions and place them on the first line inside the function
  + Do not indent docstrings for module documentation
  + Viewable using the help(*module\_name*) function
* Importing modules
  + import *module­\_name* [as *namespace*]
    - Optional to specify a *namespace* when importing modules
    - Default namespace is the name of the module
    - Example
      * import temperature as temp
      * Then use “temp” instead of temperature when accessing module functions
  + Importing into the global namespace (NOT best practice)
    - can run into naming conflicts creating a name collision
    - if you’re going to do this, you can import single functions from a module to reduce the chance of name collisions (but still not best practice to use global namespace)
    - Syntax
      * from *module\_name* import *function\_name1*[, *function\_name2…*]
      * from *module\_name* import \* # imports all of the functions (BAD practice)
* Using module functions
  + *namespace*.*function\_name*([*args*])
    - Example using example above having imported temperature as temp
      * temp.to\_celsius(32)
  + for globally imported functions, just call the function without use of a namespace
* Standard modules
  + Provided by python that include many useful functions
  + Import and use the functions like any other module
  + *module\_name*.*method\_name* # syntax for use
  + Standard modules
    - math # general math stuff
    - random # generating random numbers
    - decimal # decimal numbers
    - csv # working with csv files
    - pickle # persistent data storage
    - tkinter # building GUI apps
* Random module # random.*method\_name*
  + .random()
    - returns a random float >= 0.0 and < 1.0
  + .randint(*min, max*)
    - returns random int value >= min and <= max
    - useful for dice rolls (can do multiple rolls for multiple dice, stored in separate variables then print a result)
  + .randrange([*start*,] *stop* [,*step*])
    - returns random value >= *start*, < *stop*, and a multiplel of *step* (default step is 1)
    - useful for getting even or odd numbers (step = 2 and start should be either even or odd)
  + .choice(*list*)
    - returns a randomly selected item from the list
  + .shuffle(*list*)
    - shuffles the list randomly

**File Input/Output (I/O)**

* open(*file[, mode]*)
  + returns a file object for the specified file with the specified mode
    - Default mode is read
  + modes
    - r # read. if file doesn’t exist, throws an exception
    - w # write. if the file doesn’t exist, it creates it. if file exists, all data is overwritten
    - a # append. if the file doesn’t exit, it creates it. if the file exists, data is appended
    - b # binary. use for binary files along with r or w mode
* Opening/closing files manually
  + Bad practice to do this, because file remains open if an error occurs between open and close cmds
  + *var\_name =* open(”file\_name.txt”, “w”)

*var\_name.*write(“Test”) # writes “Test” to the file

*var\_name*.close() # closes the file, necessary to free up memory, always close files

* With Statements
  + Best practice to use a with statement because it automatically closes files when finished
  + with open(*file, mode*) as *var\_name*:

statements…

* Methods of File Objects
  + write(*str*) # writes the specified string to the file, must include \n if you want to start on a new line or end with new line char
    - When using print() you may have to specify end=”” if a newline character exists at the end of each line
    - for line in file:

print(line, end=””) # removes the \n char from the print cmd

* + read() # reads the entire file and returns its contents as a string
  + readlines() # reads the entire file and returns it as a list
  + readline() # reads the next line in the file and returns it as a string
* Text Files
  + Writing Strings
    - *list\_name =* […list contents…]

with open(“*file\_name.txt”*, “w”) as file: # could also use “a” to append

for i in *list\_name*:

file.write(i + “\n”) # stores each list item on a new line

* + Reading Strings from a file (containing a list) and storing into a list
    - *list\_name* = []

with open(“*file\_name.txt”*) as file:

for line in file:

line = line.replace(“\n”, “”) # removes new line char, if items on new lines

*list\_name*.append(line)

* + Writing Numbers to a text file
    - *list\_name* = […list contents…]

with open(“*file\_name.txt”*, “w”) as file:

for i in *list\_name*:

file.write(str(i) + “\n”) # converts to strings

* + Reading Integer Numbers from a file (containing a list) and storing into a list
    - *list\_name* = []

with open(“*file\_name.txt”*) as file:

for line in file:

line = line.replace(“\n”, “”) # removes new line char, if items on new lines

*list\_name*.append(int(line)) # converts string to integer

* **CSV Files**
  + Python provides a CSV module
    - import csv
  + Writing CSV Files
    - **Remember to always include this stuff in functions (def write\_file(*list\_name*))**
    - Add a third arg to the open() function to enable “universal newlines mode”, to read/write new lines correctly for all operating systems
    - A writer function is needed to return a CSV writer object, then a method of this object is used to write data to the csv file
    - Example
      * *list\_in2d* = [[…first row data…], […second row data…]…] # 2d list, each row a list

import csv # import csv module

with open(“*file\_name.csv”*, “w”, newline=””) as file: # using universal newlines

*writer* = csv.writer(file) # create CSV writer object (named *writer* here)

*writer*.writerows(*list\_in2d*) # use writerows(*list*) method to write the file

* + Reading CSV Files
    - reader(*file*)
      * returns a CSV reader object with the csv data from the file
    - Example
      * with open(*“file\_name.csv”*, “r”, newline=””) as file

*reader =* csv.reader(file)

* + - * Best practice to include this in a function
        + in the function, create a new empty list
        + in the “with statement”

create the reader object

**use to generate your list**

for row in *reader\_object*:

*list\_name*.append(row)

* + - * The data are stored in a list that can be processed using for statements etc.
        + the for statement (for *i* in *reader*) the *i* represents each row

so often (for row in *object\_name:*)

each column is a list item (row[0] to access first item in a row)

this is if you don’t use the method above for creating a list

* + Changing the CSV format
    - “quoting” argument
      * specifies when quotes are written and read
      * helps when data contain single or double quotes
      * quoting=csv.QUOTE\_MINIMAL # quote\_minimal is the default
        + quote minimal only adds extra quotes to columns that contain special chars (like delimiters, quotes, or end of line chars) when writing files
      * quotechar=’”’
        + specifies the char used to quote columns, default is double quote
      * delimiter=”,”
        + specifies a one-char string used to separate fields (default is comma)

can also use “\t” for tab delimited

works reading/writing files

* **Binary Files**
  + Use the “pickle module” to work with binary files (comes standard)
  + Writing Files
    - called serialization
    - dump(*object, bfile*) # writes the specified object to the binary file
    - Example
      * import pickle

with open(“*file\_name.bin*”, “wb”) as file: # use “wb” for write binary

pickle.dump(*list\_name*, file) # writes your list to the file

* + Reading Files
    - called deserialization or unpickling
    - load(*bfile*) # reads an object from the binary file
    - Example
      * import pickle

with open(“*file\_name.bin*”, “rb”) as file: # “rb” for read binary

*list\_name* = pickle.load(file) # stores contents of the file in a list

**Exception Handling**

* Use “try statements” to catch exceptions
* Try statement syntax
  + try:

*statements*

except [*ExceptionName*]:

*statements*

[except [ExceptionName]:

*statements*

[finally:

*statements*]

* + Can include multiple except with exception types (see multiple exceptions)
  + Either use a “with statement” to clean up resources when done or include a finally statement to free resources (like file.close())
  + **Finally** statements **are always executed** even if the try statement works or an exception is handled
* **Handling one type of exception (like a ValueError)**
  + Best practice to be specific like this if possible or see multiple exceptions section
  + try:

*num* = int(input(“Enter an integer: “)) # try to get user to enter an int

except ValueError: # this only accepts a ValueError exception

print(“You entered an invalid integer. Please try again.”)

print(“Thanks!”)

* Handling all types of exceptions
  + Best practice to handle a specific exception and not use “except:” or see multiple exceptions section
  + try:

*num* = int(input(“Enter an integer: “)

except: # accepts any type of exception

print(“Invalid entry, please try again.”)

print(“Thanks!”)

* **Multiple Exceptions**
  + Hierarchy for exception classes
    - Exception
      * OSError
        + FileExistsError
        + FileNotFoundError
      * ValueError
  + Code multiple exception types from most specific to least specific using syntax below
  + Syntax # using open file as example to show multiple levels
    - **using the sys module to define an exit\_program function**
    - import sys # import sys module

def exit\_program: # the exit\_program function

print(“Terminating Program.”)

sys.exit()

FILENAME = “*file\_name.csv”* # set global variable FILENAME

try:

with open(“FILENAME”*,* “r”, newline=””) as file: # “r” not necessary

reader = csv.writer(file)

for row in reader:

*list\_name*.append(row)

return *list\_name*

except FileNotFoundError:

print(“Could not find “ + FILENAME+ “ file.”)

exit\_program() # exits the program if doesn’t work (instead of while True:)

except Exception as e:

print(type(e), e)

exit\_program()

# if you didn’t use the “with” statement, you would add

finally: file.close()

* Using While Statements within a function in conjunction with exception handling for user inputs
  + Example
    - def *function\_name*():

while True:

try:

*var\_name* = float(input(“*Input text:* ”)) # float as example

return *var\_name* # ends while loop when executed

except ValueError: # best practice to specify error type

print(“Invalid entry. Please try again.”) # text to display, retries loop

continue # may need a continue statement here

# code keeps searching for inputs until the return statement is executed

* Raising (throwing) exceptions
  + forcing an exception to occur
  + Syntax
    - raise *ExceptionName*(“*Error message*”) # ExceptionName could be ValueError etc.
  + Good practice to raise exceptions in functions used by other programmers to validate function args
    - Example
      * if len(*var\_name*) == 0:

raise ValueError(“The *var\_name* argument is required.”)

* + Remember to **remove certain raised exeptions after testing is complete** (ones for testing purposes)
  + Common to create a **log\_exception(e)** function to put inside your except clause, but then you need to raise e again so the function can handle the exception
* Big Exception Example
* def add\_movie(movies):  
   name = input("Name: ")  
   while True:  
   try:  
   year = int(input("Year: "))  
   except ValueError:  
   print("Year must be a valid integer greater than 0")  
   continue  
   if year <= 0:  
   print("Year must be greater than 0.")  
   continue  
   else:  
   break
  + Using continue and break allows add\_movie to be a larger function that includes more code after the while statement. Only necessary when when in a loop. (both continue statements in the while loop)
* Creating Custom Exceptions # uses inheritance described in the OOP section below
  + Use inheritance to create a new type of exception
  + Hierarchy for six common exceptions
    - Exception
      * NameError
      * OSError
        + FileExistsError
        + FileNotFoundError
      * ValueError
  + Creating a custom exception
    - Syntax
      * class *CustomErrorName*(*ExceptionClassName*):

pass

# a single pass statement will make this exception work just like the one it inherits

* + - * This code creates a *CustomErrorName* exception that has all of the properties and methods inherited from the *ExceptionClassName* that already exists
    - Example
      * class DataAccessError(Exception):

pass

* + - * The DataAccessError exception is treated just like an exception
  + Using a custom exception (see p. 425 Murach for more details)
    - Need to be sure to import the *ExceptionClassName* from its module
      * from *moduleName* import *ExceptionClassName*
    - use *try:/except ExceptionName:* with your custom *ExceptionName*
    - Example
      * try:

*statements*

except *ExceptionName*:

raise *CustomExceptionName*(“*Informative text about the error.*”)

except Exception:

raise *CustomExceptionName*(“*Slightly different informative text*.”)

* + - * you can basically convert exception types (*ExceptionName*) into your custom exception
      * you can catch different exception types including a catchall Exception and have different error messages for each type

**Object Oriented Programming**

* Unified Modeling Language (UML) class diagram is a graphic organizer used to show the attributes and methods of one or more classes
* OOP is usually done using **camelCase** (class names usually have each word capitalized)
* Classes can be used to created similar object types
  + These can store data in ***attributes*** and can also have ***methods*** that are usually written **camelCase**
  + Use “self” to access attribute and methods (rather than “this” like JS)
    - **All methods including the constructor must accept an arg of self (“could be ‘this’ but ‘self’ by convention in Python)**
  + Attributes/methods can be public or private, and the code in “defining classes” creates public ones
    - Code outside the class can get/set these values when they are public
  + Common to use modules to store class info
    - then import the module or just the class(es) you want to use
      * Example
        + from *module\_name* import *ClassName1*[, *ClassName2*] …
* Defining Classes
  + Syntax
    - class *ClassName*:

# a constructor method (always named \_\_init\_\_(*args*)) used to initialize

# this function is automatically called whenever an object is created from this class

def \_\_init\_\_(**self**, [*initArg1*][, *initArg2*]…): # *initArgs* are used to set attrs.

self.*attr1 = initArg1* # attr1 and arg1 usually the same (i.e. “name”)

self.*attr2 = initArg2* # and so on

# can set default values for attributes by supplying *initArg1=value*

# values can be strings, int, float, etc. and code accordingly

# use named args when creating an object from classes with default values

# **good practice to override the \_\_str\_\_() method of the object class (section below)**

# code any methods that objects in this class should contain (usually in camelCase)

def *methodOne*(**self**[, *parameters*]): # must code self, and can supply other args

# self.*attr1* must use “self” to refer to attributes within these methods

# methods can reference other methods

def *methodTwo*(**self**):

return self.*attr1* - self.*methodOne*()

* Creating Objects Using a Constructor
  + *objectName* = *ClassName*(*args*) # camelCase usually used for OOP
    - self is not passed when doing this, so if there are 4 total args (self + 3 values) you just code the 3 values needed
  + Used named args (*initArg1=value*) whenever your \_\_init\_\_() function has default values
* Accessing attributes and methods
  + *objectName*.*attributName* # for attributes (usually name in camelCase)
  + *objectName*.*methodName*([*args*]) # for methods (usually name in camelCase)
  + Remember that these return values that can be formatted etc.
  + Example
    - print(“*Title of Value*: {:.2f}”.format(*objectName*.*methodName*()))
* Public Attributes and Methods
  + Setting Attributes
    - *objectName.attributeName* = *value* # will set the value of the attribute
  + Getting Attributes
    - *var\_name* = *objectName.attributeName* # stores the value in a variable
  + Calling Methods
    - *var\_name* = *objectName.methodName*([*args*]) # calls and stores the value in a variable
* Private Attributes
  + Also called *Encapsulation* or *Data Hiding*
  + Other code outside the object can’t access these attributes
    - Must code *public methods* or *properties* (a special type of method) that are an indirect way to access the data
  + Use double underscore to create private attributes
    - self.\_\_*privateAttributeName* = *value*
    - value could be an *initArg* or a supplied value within the constructor function of the class, but would be most common to supply the value in the constructor function
  + Should supply an indirect way to access the data through a method as needed
    - def *getPrivateAttributeValue*(self):

return self.\_\_*privateAttributeName*

* + Accessing hidden attributes using **methods**
    - def *methodName*(self):

self.\_\_*privateAttributeName* = *statements that keep the attribute integrity*

# or other calculations that set self.\_\_*privateAttributeName*

* + - def *methodName*(self, *value*): # **allow users to set a value if it meets criteria**

if *value* …*criteria*… :

raise ValueError(“Value supplied is out of range, try again.”)

else:

self.\_\_*privateAttributeName* = *value*

* + Accessing hidden attributes with **properties**
    - Create properties using *annotation* (@)
    - If coded properly, you only need to code this once
      * if you code a @*methodName.*setter (where *methodName* = attribute name = parameter in the \_\_init\_\_(*parameters*) constructor)
      * when you set self.*attributeName* = *parameter* # it will pass the *parameter* to the setter function for validation (see p. 395 Murach if this isn’t clear)
    - Syntax
      * @property
        + coded above the “getter” method for the property
        + always “@property”
      * @*privateAttributeName*.setter
        + coded above the “setter” method for the property
        + uses the private attribute name, a dot, and the word setter
      * Example
        + def \_\_init\_\_(self, *privateParameterName*):

self.*privateAttributeName* = *privateParameterName*

# if *privateParameterName = privateAttributeName = methodName*

# the *privateParameter* that is passed to the constructor will be passed to the setter property for validation automatically

# the getter method

@property

def *methodName*(self): # *methodName* should be what you call the attr

return self.\_\_*privateAttributeName*

# the setter method

@*methodName*.setter # *methodName* should be what you call the attr

def *methodName*(self, *privateParameterName*);

if *privateParameterName* …criteria… :

raise ValueError(“Value out of range…”)

else:

self.\_\_*privateAttributeName* = *privateParameterName*

Example 2

class Product:

def \_\_init\_\_(self, price=0.0):

self.price = price

# passes price param to price setter just like if you typed it later

@property # the getter

def price(self):

return self.\_\_price

@price.setter # the setter

def price(self, price):

if price < 0:

raise ValueError(“Price can’t be less than 0.”)

else:

self.\_\_price = price

# using and accessing the \_\_price can be done by using it as a public attribute without changing the interface

product1 = Product() # can supply a valid price in () if you want

print(product1.price) # prints 0.0

product1.price = -11 # ValueError

product1.price = 10 # sets the new value for \_\_price

* + - This allows you to use the same method name for the getter/setter methods
      * so you have two functions with the same name and a diff # of args
      * the getter only accepts (self)
      * the setter accepts (self, *value*)
        + you use validation like with “accessing hidden attributes with methods” section above
      * **the method name should basically be the attribute name**
        + then you can get/set the same way as a public attribute

*objectName.methodName* # would return the current value

*objectName.methodName* = *value* # would set the value if validated

* **Inheritence (subclasses)**
  + You can create a subclass that **inherits the public attributes and methods** of its superclass
    - subclasses can add additional attributes and methods or **override the attributes/methods** inherited from the superclass
    - Example
      * Product (superclass, base class, or parent class)
        + Movie and Book (subclasses, derived classes, or child classes)
  + The UML diagram has an arrow with an open point (the triangle end of the arrow) going from the subclass to the parent class
    - the attributes and methods of the parent class are shown
    - additions or changes in the subclasses are shown there (p. 407 Murach)
  + **Defining a Subclass**
    - Syntax
      * class *SubClassName*(*SuperClassName*): # more detailed example below
        + def \_\_init\_\_(self, *parameters*): # initialize function like w/ classes

self.*attributes = parameter* # initialize attributes just like w/ classes

* + - * + def *methodName…* # define methods like w/ classes
      * You can directly access the public attributes and call public methods within the subclass code
        + You often need to call a function to create a new object within the superclass
        + **Example**

class *SublcassName*(*SuperClassName*):

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

# include all *parameters* here needed for the superclass and the subclass

# **call superclass constructor to create the object**

*SuperClassName*.\_\_init\_\_(self, *parameters*)

# set any new subclass *attributes* from *parameters*

# **good practice to override \_\_str\_\_() method**

# section on overriding \_\_str\_\_() method below

# define any new (or overriding) methods

# **should call original method if modifying**, or completely override if that’s what you want by not calling the original method

# both method names should be the same to override

def *methodName1*(self):

return *SuperClassName.methodName1*(self) + “yada yada” + self.*attributeName*

* + - Polymorhphism
      * Let’s you treat objects of subclasses as if they were the same type of object (using methods in the superclass that get overridden)
      * You write one set of code for displaying or using a method defined in the superclass, even though this was modified in subclasses
        + Different object types will execute their version of the method
* Checking Object Type
  + isinstance(*object,* **[*modName.*]***ClassName*)
    - returns True if the object is an instance of the specified class, otherwise False
    - you only need to code [*modName.*] (the module name) if you didn’t import the class directly
      * i.e. from *modName* import *ClassName*
      * **if you just imported the module, you need to code *modName.***
  + Can use the built-in isinstance() function within another function that performs different actions based on the type of object encountered (very useful when subclasses override superclass methods)
* Override Object Class Methods
  + The ***object class*** is a superclass for every single object, so every object inherits the object class
    - Sometimes you need to override methods of the object class
  + Example
    - When you supply an object as the argument of the print() method, Python calls the \_\_str\_\_() method of the object class to get information about the object
      * It’s good practice to override the \_\_str\_\_() method to provide more detailed information about the object
      * Info should be concise, informative, and easy to read
    - **Syntax for overriding the \_\_str\_\_() method**
      * def \_\_str\_\_(self):
        + return *stringForObject* # can include self.*attributes* in this description
        + # could also use ‘return self.*getDescription*()’ if you already had a function that provides a description of the object
* Define an Iterator for an Object
  + See p. 420-421 Murach for more info
  + You may need to override the \_\_iter\_\_() and \_\_next\_\_() methods for an object
    - Need to do this when your object contains other objects and you want to iterate through them
      * **This may only be necessary when the attributes you’re trying to access are private**
      * **You are creating a public way to access objects store in private attributes**
    - You will code these methods in the primary object’s methods that holds the other objects
  + Example
    - This example has an object with subobjects stored in it
      * i.e. you have a Dice object with single die objects stored in it
    - self.\_\_list = [] # create private list attribute in the container object
    - def \_\_iter\_\_(self): # override the ­­\_\_iter\_\_() method in the container object
      * self.\_\_index = -1 # initialize the index for each iteration
      * return self
    - def \_\_next\_\_(self): # override the \_\_next\_\_() method in the container object
      * if self.\_\_index >= len(self.\_\_list)-1:
        + raise StopIteration() # stops the loop if at the end of the indexes
      * self.\_\_index += 1 # increment the counter
      * *subObject* = self.\_\_list[self.\_\_index] # access your subobject stored in the list
      * return *subObject* # *subObject* is the generic name for what’s in the list (i.e. die)
    - you can then iterate through
      * for *subObject* in *objectName*:
        + *statements* referencing *subObject*.*attribute(s)*
* Creating Custom Exceptions
  + Use inheritance to create a new type of exception
  + Hierarchy for six common exceptions
    - Exception
      * NameError
      * OSError
        + FileExistsError
        + FileNotFoundError
      * ValueError
  + Creating a custom exception
    - Syntax
      * class *CustomErrorName*(*ExceptionClassName*):

pass

# a single pass statement will make this exception work just like the one it inherits

* + - * This code creates a *CustomErrorName* exception that has all of the properties and methods inherited from the *ExceptionClassName* that already exists
    - Example
      * class DataAccessError(Exception):

pass

* + - * The DataAccessError exception is treated just like an exception
  + Using a custom exception (see p. 425 Murach for more details)
    - use *try:/except ExceptionName:* with your custom *ExceptionName*
    - Example
      * try:

*statements*

except *ExceptionName*:

raise *CustomExceptionName*(“*Informative text about the error.*”)

except Exception:

raise *CustomExceptionName*(“*Slightly different informative text*.”)

* + - * you can basically convert exception types (*ExceptionName*) into your custom exception
      * you can catch different exception types including a catchall Exception and have different error messages for each type
* When to use inheritance
  + an object is a *type of* another object
    - i.e. *book* is a *product*
  + both classes are part of the same domain
    - i.e. *product, book*, and *movie* are all part of the same domain trying to define types of products
  + the subclass primarily adds features to the superclass
  + DO NOT use inheritance if it violates encapsulation allowing users to access stuff directly
* **Designing an Object-Oriented Program**
  + Five Steps for Designing an OOP
    - Identify data attributes
      * analyze existing systems
      * evaluating comparable systems
      * interviewing anyone using the system
    - Subdivide each attribute into its smallest useful components
      * parts of an address, first name & last name, etc.
    - Identify the classes
      * list the classes, and group your attributes with each class
    - Identify the methods
      * identify anything that is a calculation
      * identify anything you can use *getter properties* rather than *getter methods*
        + using encapsulation, which is probably better anyway
      * maybe useful to provide *getDataValueStr* (string versions) of numbers for display purposes
      * Keep in mind that the **public attributes, properties, and methods** define the object’s **user interface** (which is also why converting methods to properties is useful)
    - Refine the classes, attributes, and methods
  + **Three-Tier Architecture**
    - very common for structuring programs (see p. 446 Murach for an example)
    - Master Example at the end of this doc that uses the three tier method and interfaces with a database
    - The Tiers
      * Presentation (UI) Tier
        + contains the primary main function to run the application

if \_\_name\_\_ == “\_\_main\_\_”:

main()

* + - * + handles details of the user interface
        + code for this tier in a console application is typically procedural and not object-oriented
        + code for this tier in a GUI application is typically object-oriented
      * Business Tier
        + interface between the database tier and presentation tier
        + examples: products/customers, business calculations
        + includes business classes and business objects
      * Database Tier
        + code for data access required by the program
        + retrieving, adding, deleting, and updating data
        + works with data stored in databases or other files
        + common to use classes/objects or just to use functions to provide this access
  + Example: A Shopping Cart Application
    - products, line items, and shopping carts
      * the shopping cart can display information in line items when viewing the cart
      * it’s useful to create this “line item” object for this purpose

**Databases**

* Python has built-in support for SQLite
  + SQLite data types
    - TEXT
    - INTEGER
    - REAL # decimals
    - BLOB # data stored exactly as entered
  + **DB Browser for SQLite** (Murach recommended)
* **Connecting to a SQLite Database**
  + Use sqlite3 module
    - import sqlite3
  + Connect to database and return a connection object
    - conn = sqlite3.connect(*string\_path\_to\_database\_file*) # use “” because is a string
      * if database is in working dir, same dir as the program, just specify file name
      * good practice to use a variable to store the database file string
      * use .close() method of connection object to close connection once the object is created
        + see example below
    - **If database is NOT in the working dir**
      * need to provide connection route for different OS specs
        + this requires importing modules to check for system config
        + code below assumes mac/linux users store files in HOME dir
      * Example
        + import sys

import os

if sys.platform == “win32”: # use win32 for all “bit” versions of windows

*DB\_FILE* = “*file\_path*” # i.e. “/murach/python/\_db/movies.sqlite”

else: # mac os and linux

HOME = os.environ(“HOME”) # assumes file in home directory

*DB\_FILE* = HOME + “*file\_path*” # usually in Documents dir

conn = sqlite3.connect(*DB\_FILE*)

# good practice to set row\_factory attribute here (see example below)

**# if conn (connection obect) exists, close connection to free resources**

**if conn:**

**conn.close()**

* **Executing SQL Statements Using Python**
  + Get a cursor object for the database using the .cursor() method of a connection object
    - c = conn.cursor() # gets the cursor object
    - need to close the cursor when finished
      * use closing(*resource*) function from the contextlib module
      * **from contextlib import closing**
      * automatically close cursor when complete
        + with closing(conn.cursor()) as c:

*query statements*

* + - * + closes the connection whether or not an exception is thrown
  + Execute statements using the .execute(*sql*[, *params\_tuple*]) method
    - executes your statement store in the *sql* variable with optional *params\_tuple*
    - the cursor object (c from the example above) contains the result set
  + Access rows from a SELECT query
    - two methods of a cursor object (c from example above)
      * .fetchone()
        + returns a tuple containing the next row from the result set
        + returns None if there is no next row
      * .fetchall()
        + returns a list containing all of the rows from the result set
    - should store the result in a variable
      * *results* = c.fetchall()
    - can access using index # *result*[*index*]
      * can be problematic if columns are added later on, changing index values
    - **can access using name with the row\_factory attribute**
      * set the row\_factory attribute of your connection object to sqlite3.Row when you create your connection object
      * conn.row\_factory = sqlite3.Row
      * now can access using column names # *result*[“*columnName*”]
  + INSERT, UPDATE, and DELETE statements in Python
    - these statements will make changes, but changes aren’t saved until .commit() method is called on the connection object
    - within the “with closing” statement
      * conn.commit() # see example below
  + ***sql* SELECT query example**
    - with closing(conn.cursor()) as c:

query = ‘’’SELECT \* FROM *Table*

WHERE *column\_name* = ?’’’

c.execute(query, (*value*,))

* + - best practice to use three single quotes to wrap your query text
      * this way you don’t have to concatenate multiple lines of code
    - use ‘?’ to incorporate variables into your sql statement
      * you then supply a tuple as the second arg for the execute method
      * recall that a tuple with one value needs a comma at the end
        + *myTuple* = (*value*,)
      * can store tuple values in variables to and supply like example below
        + *var1 = value1*
        + *var2 = value2*
        + c.execute(query, (*var1, var2*))
  + ***sql* INSERT query example (same basic syntax as UPDATE and DELETE requiring .commit())**
    - *var1 = value1* # store data invariables to add to db

*var2 = value2*

*var3 = value3*

with closing(conn.cursor()) as c:

sql = ‘’’INSERT INTO *tableName* (*column1, column2, column3*)

VALUES (?, ?, ?)’’’

c.execute(sql, (*var1, var2, var3*)

**conn.commit() # save the db on completing**

* + - UPDATE
      * UPDATE *tableName* SET *columnName1 = ?* WHERE *columnName2 = ?*
    - DELETE
      * DELETE FROM *tableName* WHERE *columnName = ?*
* Database Exception Handling
  + try:

with closing… all the way to your *result(s) =* fetchone() or fetchall() statement

except sqlite3.OperationalError as e:

print(“Error *interfacing* *with database* -“, e)

*result(s) =* None

if *result(s)* != None

*statements to loop or handle data*

* **Looping through all rows of a SELECT query using fetchall**
  + *results =* c.fetchall() # might want to run a print(*results*) format of what is returned
  + for *result* in *results:*

*statements using result[“columnName”] or result[index]*

* Quick reminder of SQL statement order
  + *ACTION* (\* or, *column1,…[AS aliasName1…]*)

*FROM/INTO* *table\_name* or *SET (for UPDATE action)*

*[[INNER] JOIN table-2* ON *table-1.column-1 = table-2.column-2]*

WHERE

ORDER BY *column-1* ASC

**GUI Using *tkinter* Toolkit**

* *Root* window # first component
  + root window includes buttons, labels, text entry etc. # components can be called *widgets*
  + .Tk() constructor creates the root window # see example below
  + .title(*str*) method to add a title
  + .geometry(“*widthxheight*”) # specify width x height using pixels as string
    - root.geometry(“300x200”) # example
  + **.mainloop() # must call to display root window and enter *event processing loop***
    - program will not run unless this is called!
    - it just executes the code in the script in sequence then exits
    - must come after the code that sets up the window, otherwise window is displayed before setup code is executed
  + .destroy()
    - closes the root window and causes the event processing loop to end (ends the program)
    - usually called within a function that is attached to a button command
    - can also add a function to display a message to allow saving prior to exiting or displaying that all unsaved data will be lost
    - def exit\_program():
      * root.destroy()
* Components # from tkinter import ttk
  + Frames
    - A frame is a component (invisible container) used to group components (like buttons)
      * need to import the ttk module from the tkinter module
      * **from tkinter import ttk** # otherwise, would have to tk.ttk.yadayada methinks
    - Frame and Button are classes that need to be created within a parent component (like the root)
    - Syntax
      * ttk.Frame(*parent*[, *padding*])
        + creates a frame object and adds it to the specified parent component with optional padding
        + padding is in pixels, tho no units are specified, and is btw edge of frame and parent
        + Example

frame = ttk.Frame(root, padding=”10 10 10 10”)

need to frame.pack() after creating the frame (see methods below) to see it

* + - **Creating a custom Frame Class**
      * inherit the ttk.Frame then add components and event handlers to the class
      * Syntax
        + import tkinter as tk

from tkinter import ttk

class *ClassName*(ttk.Frame):

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

ttk.Frame.\_\_init\_\_(self, parent, padding=”10 10 10 10”)

self.pack(fill=tk.BOTH, expand=True)

# define variables & create entries, labels, buttons etc.

# add padding to all components

for child in self.winfo\_children():

child.grid\_configure(padx=5, pady=5)

# define callback method for a clear button

def clear(self):

self.*varBoxToClear*.set(“”)

if \_\_name\_\_ = “\_\_main\_\_”:

root = tk.Tk()

root.title(“*Title*”)

*ClassName*(root) # create the frame using your class

root.mainloop()

* + Buttons
    - ttk.Button(*parent, text*[, *command*])
      * creates a button object with the specified text and adds it to the parent component
      * a button click will call the function specified by the *command*
      * Example
        + button1 = ttk.Button(frame, text=”Click me!”, command=click\_button1)
        + need to button1.pack() to display the button # or use .grid()
        + clicking calls the click\_button1() function
  + Labels
    - ttk.Label(*parent*, *text*)
      * creates a label with the specified text and adds to the *parent* component
      * Examples
        + nameLabel = ttk.Label(frame, text=”Name:”)

nameLabel.pack() # display the label (could also .grid())

* + - * + if no variable is needed to work with the label later, chain the .pack() or .grid()

ttk.Label(*parent, text*).grid(*options*)

* + Text Entry
    - aka text box, text field, entry field
    - before creating, typically create a StringVar object to get/set the text in this field
    - **Create StringVar Object**
      * *varName* = tk.StringVar() # creates the empty object and assigns a variable
    - **Create Entry Box**
      * *varName2* = ttk.Entry(*parent, width, textvariable=varName*[, *state*])
      * *varName* should be the StringVar() object you already created
      * *width* specifies the width of the field in pixels
      * *state* can set to “readonly” so code can set, but not the user
      * .pack() or .grid() to display the box
    - Get values from entry box
      * *var* = *stringVarObjectName*.get() # use the StringVar Object name (*varName*)
    - Set values for entry box
      * *entryVar*.set(*value*) # strings should go in “”
  + Methods # all components
    - .pack([*fill*][, *expand*])
      * makes the component visible # can use .grid() instead
      * default is to auto size the frame to be just large enough to hold the components it contains (can edit with fill & expand args)
      * to auto resize vertically and horizontally to fill parent component, set fill=tk.BOTH and expand=True
      * can create and pack all at once (second example)
      * Example
        + frame.pack(fill=tk.BOTH, expand=True)
        + ttk.Label(frame, text=”Email Address”).pack() # no variable needed
    - .grid([*column][, row][, sticky][, padx][, pady][, columnspan][, rowspan]*)
      * creates a grid layout for your components within a frame
      * column # column index where the component is added (0 indexed)
      * row # row index where the component is added (0 indexed)
      * sticky # justification (tk.E, tk.W, tk.S, tk.N) for north south etc.
        + can supply a tuple for multiple directions
      * padx # horizontal padding between this component and the next
      * pady # vertical padding
      * columnspan # number of columns the component should span
      * rowspan # number of rows the component should span
    - .winfo\_children()
      * used to set the options for all components within a frame
    - .grid\_configure(*same\_options\_as\_grid\_method*)
      * used to set the options for all components within a frame
    - Example of .winfo\_children() and .grid\_configure()
      * for child in *frame*.winfo\_children():

child.grid\_configure(*set\_options*)

* + - * uses the same options above, often used to set padx and pady
      * use your *frame* name
* Ending a Program
  + common to create a button to do this that calls the destroy() method of the root window
  + button\_exit = ttk.Button(*frame\_name*, text=”Close and Exit”, command=exit\_program)
  + def exit\_program():
    - root.destroy()
* Example Main Function
  + import tkinter as tk # import the module into tk namespace

from tkinter import ttk # import the ttk module for frames/buttons

root = tk.Tk() # creates the root window and assigns to a variable

root.title(“*Application Title*”) # title for the root window, usually the app title

*ClassName*(root) # use a custom class to define the components (see above)

root.mainloop() # display the root window (after setting up the root window)

**Debugging**

* Syntax Errors
  + Usually caught when trying to run the program by an IDE
  + Common syntax errors in Python
    - misspelling
    - missing colons
    - missing “” or ()
    - misusing () & []
    - indentation
    - using keywords for variable names
    - type errors
* Runtime Errors
  + Throw exceptions
* Logic Errors
  + Don’t crash or throw exceptions, but produced undesired effects
* Testing phase
  + Test with valid data and invalid data
  + Use print() statements to print information to the console, so you can see if/when something goes wrong
  + Test as you build and do it one piece at a time
  + Test functions one at a time in the IDLE shell
  + Use the debugger in your IDE
    - view the stack

**Sample Program**

* db module (database tier)

# import database modules

import sys

import os

import sqlite3

from contextlib import closing

# import object classes

from *objects* import *ClassName*

conn = None # initialize connection object as None (global variable)

# database connection function, windows/mac/linux support, row\_factory function for using column names

def connect:

global conn # access global conn variable

if not conn:

if sys.platform == “win32”:

DB\_FILE = “*file\_path*” # set file path if windows “/murach/python…”

else:

HOME = os.environ(“HOME”) # for mac/linux

DB\_FILE = HOME + “/Documents/*file\_path*” # usually in Documents dir

conn = sqlite3.connect(DB\_FILE)

conn.row\_factory = sqlite3.Row

# closing function, no need to use global, because only accessing conn variable, not changing it

def close:

if conn:

conn.close()

# typically have code that creates objects from database results

# these are coded prior to SELECT queries, because they are used in the function containing SELECT queries

def make\_*object1*(row): # use (row) as the param, because it’s a tuple/list with values to access

return *Object1*(row[“*columnName*”][, row[*“cloumnName2*”]…)

# supply params to create object instance

def make\_*object2*(row):

return *Object2*(row[“*columnName3*”][, make\_*Objec1*(row))

# can call previous object constructors to include objects as params here

# database queries

def get\_*databaseinfo*([*arg(s)*]): # also your ADD/DELETE/UPDATE etc. queries

query = *statements*

with closing(conn.cursor()) as c:

c.execute(query)

results = c.fetchall() # or row = fetchone()

# can include make\_*object*(row) in this type of function

* ui module (presentation tier) # console driven

#!/usr/bin/env/python3

import db

from *objects\_module* import *Class(es)*

def display\_title():

print(“*Application Title*”)

print()

display\_menu()

def display\_menu():

print(“COMMAND MENU”)

*additional print statements to enumerate the menu*

print()

def display\_*databaseInfo*():

print(“*DatabaseInfo*”)

*data =* db.get\_*databaseinfo*() # run method to execute query

for *datum* in *data*:

print(str(*datum*.*attribute*) + “. “ + *datum.attribute2*) # just an example, using str() for nums

print()

def add\_*databaseinfo*():

*var1 =* input(”*Variable Name*: “)

*var2 =* input(”*Variable2 Name:* “) # collect as many vars as needed

# could also ask for a file, parse data into variables to add to the db

# for example: could use *sampleID = db.get\_sampleID*(*sampleID*) to check if a sample exists

# if *sampleID ==* None: yada yada, else statements to add the data and print success message

# also include DELETE/UPDATE etc.

def main():

db.connect()

display\_title() # this title function also calls the display\_menu() function

# maybe other display calls here

while True: # sort of like switch statements

command = input(“Command: “) # get command from user

if command == “*option1*”:

*statements to execute*

elif command == “*option2*”:

*statements to execute*

elif command == “exit”:

break # break statement ends the loop

else:

print(“Not a valid command. Please try again.\n”)

display\_menu()

db.close() # only executes when while loop is ended (exit command)

print(“Bye!”)

if \_\_name\_\_ == “\_\_main\_\_”

main()