# 11. Testing Your Code

## Testing a function

* The unittest module from python has tools to automate testing
* Unit test - verifies that one specific aspect of a function’s behavior is correct
* Test case - a collection of unit tests that together prove that a function behaves as it’s supposed to, within the range of situations you expect it to handle.
* Full coverage - a test case with a full range of unit tests covering all the possible ways you can use a function.

| import unittest  from name\_function import get\_name  class NamesTestCase(unittest.TestCase):  def test\_first\_last(self):  self.assertEqual(name, 'Dakota Gullicksen')  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main() |
| --- |

* Output works: The dot tells us a single test passed, the second line tells us it took less than a 0.001s to run, and the last line tells us that all the tests can be passed

| .  ----------------------------------------------------------------------  Ran 1 test in 0.000s  OK |
| --- |

* Output fail: The E means there was an error, next it tells us the error is the function, the trackback helps by telling us we are missing an argument, we ran the test in 0.002s, and lastly it tells us that we have one error

| E  ======================================================================  ERROR: test\_first\_last (\_\_main\_\_.NamesTestCase)  ----------------------------------------------------------------------  Traceback (most recent call last):  File "C:\Users\kotag\OneDrive - Wayne State College\python\_works\ch11\ch11test\_name.py", line 6, in test\_first\_last  name = get\_name('Dakota', 'Gullicksen')  TypeError: get\_name() missing 1 required positional argument: 'last'  ----------------------------------------------------------------------  Ran 1 test in 0.002s  FAILED (errors=1) |
| --- |

* + We can run as many tests as we want just like that function, we can even do middle names. And when we test it there will be 2 dots and it will say that it ran 2 tests in a certain time.

## Testing a class

* (look at files ch11survey, ch11language\_survey, and ch11test\_survey for examples)
* Assert methods test whether a condition you believe is true at a specific point in your code is indeed true.
  + assertEqual(a, b) - verifies a==b
  + assertNotEqual(a, b) - verifies that a !=b
  + assertTrue(x) - verifies that x is true
  + assertFalse(x) - verifies that x is false
  + assertIn(item, list) - verifies that item is in list
  + assertNotIn(item, list) - verifies that item is not in list
* setUp() - allows you to create objects once and then use them in each of your test methods. Python first runs the setUp() before any test\_. This will make the code less repetitive.

# 10. Files and Exceptions

## Reading from a File

* + When you want to work with the information in a text file, the first step is to read the file into the memory

| with open(‘file.txt’) as fileObject:  contents = fileObject.read()  print(contents) |
| --- |

* + The open() function opens that file and the argument would be the file (we’ll use .txt). It will only look for that file if it is in the same folder as the program looking for it. The open(‘file.txt’) returns an object representing the file to the fileObject variable
  + with closes, the file once access to it is no longer needed. If files aren’t closed properly it could lead to lost or corrupted data
  + The read() method reads the entire contents of the file into one long string
  + File path tells python where to look in a specific location on your system. A relative file path tells python to look for a given location in the directory where the program is stored. An absolute file path specifically describes where to find the file
    - with open(‘folder/file.txt’) as fileObject: - relative file path
    - filePath = ‘home/folder1/folder2/folder3/file.txt’ - absolute file path

| with open(‘file.txt’) as fileObject:  for line in fileObject:  print(line) |
| --- |

| with open(‘file.txt’) as fileObject:  lines = fileObject.readlines() - takes each line from the file and stores it in a list  for line in lines: - each index stores one lineof the .txt file  print(line) |
| --- |

| with open('pi\_digits.txt') as file\_object:  lines = file\_object.readlines()  pi\_string = '’  for line in lines:  pi\_string += line.strip()  print(f”{pi\_string[:52]}”) - will only print out the first 52 numbers  print(len(pi\_string)) |
| --- |

| with open('pi\_digits.txt') as file\_object:  lines = file\_object.readlines()  pi\_string = ''  for line in lines:  pi\_string += line.strip()  birthday = input("Enter your birthday, in the form mmddyy: ")  if birthday in pi\_string:  print("It seems that your birthday is in the first million digits of pi!")  else:  print("Your birthday isn't in the first million digits of pi.") |
| --- |

* + You can write through a program to save data. The second argument ‘w’ tells python we want to open the file in write mode. There are other modes like ‘r’(read), ‘a’(append), and ‘r+’(read and write). The thing with write mod if the file does exist it will erase all of the contents. The write() allows us to write a string into the fileObject

| filename = 'programming.txt'  with open(filename, 'w') as fileObject:  fileObject.write("I love programming!")  filename = 'programming.txt'  with open(filename, 'w') as fileObject:  fileObject.write("I love programming!")  fileObject.write("I love creating new games!") - adds a new line of text into the .txt file |
| --- |

* + Instead of erasing the whole file and rewriting lines in it, we can append lines into the file. If you continue playing this over and over it will add the same lines into the .txt file

| filename = 'programming.txt'  with open(filename, 'a') as fileObject:  fileObject.write("\nI love programming even more when it makes sense!")  fileObject.write("\nI love creating new programs!") |
| --- |

## Exceptions

* Exceptions - special objects that manage errors that arise during the program’s execution
* Traceback - a report of the exception that was raised
  + Exceptions are handled with try-except blocks - they will continue to do something even if something goes wrong. So instead of tracebacks users can read easy-to-read error messages.
  + Try-except-else block - python attempts to run the code in the try block, if it is successful it goes to the else block, if it fails it is caught by the except block

| try:  answer = int(fNum)/ int(sNum)  except ZeroDivisionError: - prints statement when dividing by 0  print("You can't divide by zero!")  else: - if there are no errors it prints the answer statement  print(answer) |
| --- |

* + There could be errors where there are missing files. The encoding=’utf-8 is needed when your system’s default encoding doesn’t match the encoding of the file being read

| filename = 'alice.txt'  try:  with open(filename, encoding='utf-8') as fileObject:  contents = fileObject.read()  except FileNotFoundError: -prints statement when a file isn’t found  print(f"Sorry, the file {filename} does not exist.") |
| --- |

* + We can also count words in the file using the string split() that turns the words of a string into a list. You can even make a method out of the try-catch to count multiple words in a file, just pass in the filename. Or if you have a list of files you can make a loop that goes through the list and print each response.

| filename = 'alice.txt'  try:  with open(filename, encoding='utf-8') as fileObject:  contents = fileObject.read()  except FileNotFoundError:  print(f"Sorry, the file {filename} does not exist.")  else:  words = contents.split()  numWords = len(words)  print(f"The file {filename} has about {numWords} words in it.") |
| --- |

* + You can fail silently using the word pass in except. This won’t print out anything, it tells the program to do nothing if this happens.

| except FileNotFoundError:  pass - nothing happens if there is an error |
| --- |

* + Giving users of the program information that they aren’t looking for decreases the use of the program

## Storing Data

* + A simple way to store to save the information by users is using the JSON module. The JSON module allows you to dump simple python data structures into a file and load the data from that file the next time your program runs. You can use the module to share data between different python programs or even other people not programming in python
  + json.dump() - store the set of numbers, it takes 2 arguments: the data to store and a file object to store the data

| import json  numbers = [2,3,5,7,11,13]  filename = 'numbers.json'  with open(filename, 'w') as f: - the file is in write mode  json.dump(numbers, f) - we store the list of numbers in the .json file |
| --- |

* + json.load() - reads the list back into memory, it has one argument: file object

| import json  filename = 'numbers.json'  with open(filename) as f: - we only need to read the file  numbers = json.load(f) - loads the information in the file to numbers  print(numbers) |
| --- |

* + Refactoring - breaking up the code into a series of functions that have specific jobs

# 9. Classes

* Class - represent real-world things and situations
* Objects are based on classes
  + Instantiation - making an object from a class, you work with instances of a class

| class ClassName:  def \_\_init\_\_(self, para1, para2):  self.attr1 = para1  self.attr2 = para2 |
| --- |

## Methods

* \_\_init\_\_(self, para1, para2) - a special method that python automatically runs whenever we create a new class. Sets it so the class methods can use the args that are called (basically a constructor)
  + Self - is required and must come first and will be pasted automatically so we don’t need to pass it. Each variable in \_\_init\_\_ has the prefix self, any variable that is initialized with the prefix self can be used in the class methods and can be used outside of the class
  + Attributes - variables that are accessible through instances like
    - self.attr1 = para1
  + Any other methods that don’t need any more information in the class will only have one parameter
    - def method(self):
* Making an instance from a class - this example

| class ClassName:  def \_\_init\_\_(self, para1, para2):  self.attr = para1  self.attr = para2  def classMethod(self):  object = ClassName(arg1, arg2) - this runs the \_\_init\_\_() method  object.classMethod() - calls on a class method  print({object.attr1}) - can access the class attr1 variable that was set in the class |
| --- |

* + You can create multiple instances but the object variable has to be different. You can use them the same way as before just make sure that you are calling on the right methods with the object you are using

| myObject = ClassName(arg1, arg2)  yourObject = ClassName(arg3, arg4) |
| --- |

## Working with classes and instances

* + You can use classes to represent many real-world situations. You can modify the attributes of an instance directly or write methods that update attributes in specific ways - prints out the return statement

| class ClassName:  def \_\_init\_\_(self, para1, para2):  self.attr1 = para1  self.attr2 = para2  def classMethod(self):  return variable  object = ClassName(arg1, arg2)  print(object.classMethod()) |
| --- |

* + Setting a default - attributes that don’t need to be passed in but instead can be assigned a default value

| class ClassName:  def \_\_init\_\_(self, para1, para2):  self.attr1 = para1  self.attr2 = para2  self.attr3 = value |
| --- |

* + There are several different ways to modify attribute values

| object.attr = new value - directly  object.updateMethod(value) - through a method if the method is created in the class  …  def incrementMethod(self, para): - increments the attribute  self.attr += para  …  object.incrementMethod(value) |
| --- |

* Inheritance - using another class to help define a different class. When one class inherits from another it takes its attributes and methods from the first class. The original class is called the parent class while the new class is called the child class
  + The \_\_init\_\_() for the child class

| class ChildClass(ParentClass):  def \_\_init\_\_(self, para1, para2): - all of the same parameters  super().\_\_init\_\_(para1, para2)  object = ChildClass(arg1, arg2)  print(object.ParentClassMethod()) - that is now technically also the child class method |
| --- |

* + super() - a function that allows you to call the function from the parent class
  + Once the child class inherits all of the attributes from the parent class you can add any new methods or attributes to make it different
  + To override a method make a new method in the child class that's the same name as the method in the parent class. When the method is called on through the child class method will act, not the parent class one
  + If you notice that a class has many specific attributes you can break that up into a separate class and have the class that had the attributes call on this new class

| class Class:  …statements…  self.object = NewClass  class NewClass:  …statements…  def method():  object = Class() - different object name  object.NewClass.Method() |
| --- |

## Importing Classes

* Importing classes - like importing specific methods in a file

| from module import Class - this allows us to access all the methods and variables in the Class  Or if there are multiple classes in a file  from module import Class, Class2, …  Or you can just import the module and access the classes using dot notation  import module  variable = module.Class() |
| --- |

* + Or the least recommended approach is importing all classes from the module because it’s unclear what classes you are calling on from the module and there could be naming conflicts
    - from module import \*
  + You can import your classes over several modules to keep one file from growing too large and to avoid unrelated classes in the same module
  + Like functions in chapter 8, we can use aliases when importing classes
    - from module import class as CL
  + Python itself already has some modules made
* More module information
  + https://pymotw.com/

# 8. Functions

* Function - named blocks of code that are designed to do one specific job
  + Function definition - tells the name of the function and the information needed for it. The indented lines make up the body for it.
  + Function call - tells python to execute the code in the function.

| def function\_name(parameter): - parameter is a variable that requires a value for the function to use  statement - body  function\_name(argument) - argument is the value that goes to the function |
| --- |

| def function\_name(): - empty function  statement - body  function\_name() - calls function |
| --- |

## Passing Arguments

* Positional arguments - arguments that need to be in the same order as the parameters are written

| def function\_name(parameter1, parameter2):  statement  function\_name(‘argument1’, ‘argument2’)  function\_name(‘argument3’, ‘argument4’) |
| --- |

* Keyword arguments - arguments that are name-value pair - ‘argument2’ will now be associated with parameter1 and ‘argument1’ will now be associated with parameter 2

| def function\_name(parameter1, parameter2):  statement  function\_name(parameter2 = ‘argument1’, parameter1 = ‘argument2’) |
| --- |

* Default values - the value associated with a variable that doesn’t have a value assigned to it when called - when the function is called the default for parameter2 is already assigned. If we call the function again with ‘argument3’ for parameter2 the default will be ignored. The default needs to be the last on the function call

| def function\_name(parameter1, parameter2 = ‘argument2’):  statement  function\_name(‘argument1’) |
| --- |

* + You can also use an if statement in the function to determine if there is a value assigned to it.

| def function\_name(parameter1, parameter2 = ‘ ’):  if parameter2:  statement  else:  Statement  function\_name(‘argument1’) |
| --- |

* + You can use all of these function calls at the same time.
* Return Values - a value a function returns

| def function():  variable = value  return variable |
| --- |

* + You can also use an if statement in the function to determine if there is a value assigned to it.

| def function\_name(parameter1, parameter2 = None):  if parameter2:  statement = something  else:  statement = different  return statement  variable = function\_name(‘argument1’) |
| --- |

* + Returning a dictionary

| def function(para1, para2):  dictionary = [‘key1’: para1, ‘key2’: para2}  return dictionary  variable = function(‘arg1’, ‘arg2’) |
| --- |

* Passing a list - passing a list into a function.

| def function (paralist):  for value in paralist:  statements involve a value  list = [‘thing1’, ‘thing2’, ‘thing3’]  function(list) |
| --- |

* + You can also edit lists in a function as long as the variable is declared before the function is called - after the function list2 should have something in it and list should be changed

| def function (paralist, paralist2):  for value in paralist:  statements change things from list to list2  list = [‘thing1’, ‘thing2’, ‘thing3’]  list2 = [ ]  function(list, list2) |
| --- |

* + If you don't want to empty the list use [ : ] to copy - after both lists should be filled

| def function (paralist, paralist2):  for value in paralist: - statements copy things from the list to list2  list = [‘thing1’, ‘thing2’, ‘thing3’]  list2 = [ ]  function(list[ : ], list2) |
| --- |

* Passing an Arbitrary Number of Arguments - when you don’t know how many arguments will be passed through - can receive any number of arguments for the function and can use

| def function(\*paras):  statements  function(‘thing’, ‘thing2’, ‘thing3’) |
| --- |

* + Using each argument print example - kind of like a list

| def function(\*paras):  for para in paras:  print(para)  function(‘thing’, ‘thing2’, ‘thing3’) |
| --- |

* + If you are going to pass in multiple arguments and parameters the program will first run positional, keyword, then everything else. Thing3 is associated with para1, everything else following that is part of \*para

| def function(para1, \*paras):  statement  function(‘thing3’, ‘thing’, ‘thing2’, ‘thing3’) |
| --- |

* + Keyword arguments - \*\*keyword causes python to make an empty dictionary for the function can use. You can use the dictionary normally in the function

| def function(\*\*paras):  Statement  function(key = ‘value’, key2 = ‘value2’) |
| --- |

* Storing Your functions in Modules - You can store your functions in a separate module and then import that module into your main program. Import statement tells python to make the code in the module available to the main file.
  + A module is a file ending in .py that contains the code you want to import.
  + For example a module file is made with function statements that the main (below) file can use
    - the module calls the function from the module file to be used in this file

| import module  module.function() |
| --- |

* + To import specific functions

| from module import function  Or  from module import function1, function2, … |
| --- |

* + Giving your function you’re importing an alias if a function of the same name exists
    - from module import function as an alias
  + Giving your module an alias
    - import module as an alias
  + Importing all functions
    - from module import \*

# 7. User Input and While Loops

## Input

* The input() function pauses the program and waits for the user to enter some text.
  + userInput= input(“What is your name? “) - this prints out the prompt, in this case, “what is your name?” but the userInput variable will only save whatever the user put in
    - print(userInput) - prints whatever the user typed for a name
* For a longer prompt
  + prompt = “something ”
    - prompt += “\nsomething more: ”
    - userInput = input(prompt) - will print the whole prompt before asking the user for information. In this case, the prompt will take 2 lines
* For number inputs
  + numInput = input(“How old are you? “)
    - numInput = int(numInput) - the int() makes it so the user's response changes from a string to a number, this only works from number responses
* Modulo Operator % - when % is between 2 numbers it returns the remainder. You can use this in if statements for condition test
  + 4 % 3 = 1
  + 5 % 3 = 2
  + 6 % 3 = 0
  + 7 % 3 = 1

## While loops

* While loops - run as long or while a condition is true
  + while condition\_test: - will end when the condition test is false
* Flag - a variable that determines whether or not an entire program is active. We use this if there are multiple things that would end the program
* You can use a break to exit the while loop. Put in the word ‘break’ inside the while loop and it will exit out of that loop. No matter what the condition statement is it will end the loop. (you can also use this in a for loop)
* You can use a continue to return to the beginning of the loop. (You can use this in for loops as well)
* Be careful of infinity loops as those will run forever
* Using while loops with lists and dictionaries allow you to collect, store, and organize lots of input to examine and report on later.
  + while ‘thing’ in list: - will continue to loop as long as the specific ‘thing’ is in the list

# 6. Dictionaries

## Dictionaries

* Dictionaries - allow you to connect pieces of related information.
* Dictionary - a collection of key-value pairs. Each key is connected to a value, you can use that key to get the value associated with it. The value can be a number, string, list, or another dictionary
* Key-value pair - a set of values associated with each other, you can have as many as you want
  + dictionary = {‘thing’: value, ‘thing2’: value}
* To get the value
  + dictionary [‘thing’] - this will return the value of ‘thing’
* Adding key-values
  + dictionary[‘new\_thing’] = value
  + So now the variable will have the order of {‘thing’: value, ‘thing2’: value, ‘new\_thing’: value}
* Empty dictionary
  + dictionary = {}
* Editing key-values
  + dictionary[‘thing’] = new value - now the key value when called will return the new value
* Removing key-values
  + del dictionary[‘thing2’] - this permanently removes the specified key-value
  + So now the variable will have the order of {‘thing’: value, ‘new\_thing’: value}
* get() method - use this method to set a default value that will be returned if the requested key doesn’t exist.
  + dictionary.get(‘thing3’, ‘null’) - null is the default value the program returns if ‘thing3’ isn’t a key-value in the dictionary

## Looping through a key-value dictionary

* for key, value in dictionary.items(): - for every key value in dictionary.items() (which returns a list of key-value pairs. You can change the key-value name but the first name is always the key and the second name is always the value
  + - print(“{key}”) - display the key in the dictionary
    - print(“{value}”) - displays the value associated with the key
* .keys() - when you don’t need to work with all of the values in the dictionary
  + for key in dictionary.keys():
    - print(key) - returns the key of the dictionary
* You can use sorted in the for a loop when sorting by keys
  + for key in sorted(dictionary.keys()):
* .values() - same thing as keys but instead returns the values instead
  + for value in dictionary.values():
    - print(value) - returns the value of the dictionary
* To see the values not be repeated use the set method in the for loop

| for value in set(dictionary.values()):  print(value) - prints the use only once, so if two keys have the same value the value will only be shown once |
| --- |

* + You can also set directly using braces
    - set = {‘thing’, ‘thing2’, ‘thing3’, ‘thing’}
    - // for loop
    - // output = {‘thing’, ‘thing2’, ‘thing3’}

## Nesting

* Nesting - setting dictionaries in a list, a list of items in a dictionary, or a dictionary inside a dictionary
  + List of dictionaries

| dictionary1 = {...}  dictionary2 = {...}  dictionary3 = {...}  list = [dictionary1, dictionary2, dictionary3] |
| --- |

* List in a dictionary - you can have multiple lists in a dictionary along with other key-values, basically, a key would have multiple values

| dictionary = {‘list’: [‘thing’, ‘thing2’], ‘key’: ‘value’}  for thing in dictionary[‘list’]:  print(thing)  print({dictionary[‘key’]}) |
| --- |

* Dictionary in a dictionary
  + dictionary = {‘dic1’: {‘key1’: ‘value’, ‘key2’: ‘value’}, ‘dic2’: {‘key1’: ‘value’, ‘key2’: ‘value’}}

| for dic, dicValue in dictionary.items():  print(dic)  variable = f”{dicValue[‘key1’]} {dicValue[‘key2’]}”  print(variable) |
| --- |

# 5. If Statements

* An if statement is used when the condition test (example variable == value) is true it runs the next set of statements, if not it skips the statements. The else statement is if the initial condition is false this set of statements runs.

| if condition\_test:  Statement  else:  Statement |
| --- |

* The variable can run methods in the condition test, but it doesn’t have to be there.

| if variable.method() condition\_test:  Statement |
| --- |

* if-elif-else - if the first condition doesn’t work the second if (or the elif) tests the condition, there can be multiple elif statements, sometimes you don’t need an else statement

| if condition\_test:  statement  elif condition\_test:  statement  else:  statement |
| --- |

## Condition statements

* == - equality operator returns true if both sides of the operator equal each other
* != - inequality operator returns true if both sides of the operator don’t equal each other
* > - if the left is greater than the right it is true
* < - if the left is less than the right it is true
* >= - if the left is greater than or equal to the right it is true
* <= - if the left is less than or equal to the right it is true
* and - put between separate conditions if you want more than one variable to determine whether the statements below run

| if variable == value and variable2 = value:  statement |
| --- |

* or - put between separate conditions if it doesn't matter if all or one is true for the statements below to run

| if variable == value or variable2 = value:  Statement |
| --- |

* Value in a list - if ‘value’ is in a list then true
  + if ‘value’ in list:
* Value not in a list - if ‘value’ is in a list then true
  + if ‘value’ not in list:
* Checking for empty lists
  + if list:
* Boolean - a true or false variable

# 4. Working with List

## For Loop

* For loop - a loop that goes on until it reaches a certain point, the statement repeats over and over. The best way for a computer to automate repetitive tasks. The for loop ends after the value reaches the end of the list value. Every indented line under the for loop is part of the loop. To make the program do something outside of the loop simply untab the statement. If the loop goes through the list and calls on the value outside of the list it will use the last value called. Don’t forget the colon ( : ).

| for value in list:  Statement |
| --- |

| for cat in cats:  print(f“meow, I’m {cat}”) |
| --- |

## Numerical list

* range() method - makes it easy to generate a list of numbers. The code prints 1 through 4, it stops before 5. If you put one number in the parenthesis it will start at zero and end at the number less than the number put in.

| for value in range(1, 5):  print(value) |
| --- |

* You can also put the range() method in the list() method parenthesis to help make a list variable
  + Numbers = list(range(1, 6)) - numbers now stores 1, 2, 3, 4, 5 in the list
* You can also step-size the list by inputting the third number in the range() method
  + range(2, 11, 2) - will only do the even numbers
* \*\* means the exponent of
  + Value \*\* value2 - value to the power of value2
* min(list) - returns the smallest number of the list
* max(list) - returns the biggest number of the list
* sum(list) - returns the sum of all the numbers together
* List comprehensions - allows you to generate the same list on just one line of code.
  + Squares = [value\*\*2 for value in range(1,11)] - now the squares value is a list you can use. No colon needed

## Part of a list

* list[s : l] - s is the starting index and l is the last index, returns s to l-1
  + players[0:3] - player 0, 1, 2
  + list[:l] - starts at index 0 and ends at the 3rd index
  + list[s:] - starts at s and ends at the list index of the list
  + list[-s:] - prints the last s numbers of the list
* Looping through slice
  + for player in players[0:3]
    - print(player)
* Copying a list
  + list\_2 = list[:] - this copies the entire list to list\_2
  + Simply making the list equal without the brackets doesn’t work
* Tuples - immutable list, they use parentheses instead of brackets. You basically can’t change anything in this list. If your tuple has one element there needs to be a trailing comma in the parentheses. If we want to change the tuple we need to make the variable list equal a new set of parentheses

# 3. Introducing Lists

## List

* List - a collection of items in a particular order
  + [ ] indicates a list
  + You can access any item on that list by its index, the index starts at 0
    - List = [‘a’, ‘b’, ‘c’, ‘d’]
    - List[0] - returns ‘a’
    - List[-1] - returns ‘d’, -1 returns the last element of a list, -2 returns the second to last, and so on…
    - You can then use string, char, or number methods
  + You can change an index of a list by making that index equal to something else
  + List.append(‘e’) - adds the something to the list
  + List.insert(1, ‘f’) - inserts ‘f’ at index 1 and moves the elements that were indexed at >= to 1 are shifted to the right
  + del List[1] - deletes the element at index 1 which is ‘f’ and everything shifts over respectfully
  + List.pop() - removes the last thing on the list and can return a value if it is set to equal another variable
    - Or List.pop(2) - can remove and return ‘c’ from the list
  + List.remove(‘d’) - removes the element that has the value ‘d’ and can be worked with the value
  + List.sort() - sorts the list lower case alphabetically
    - Or List.sort(reverse=True) - sorts the list in a reverse order
  + sorted(List) - sorts the list but the original order of the list is still the same, basically does change the order of the list but yet presents an ordered version of it. Can also accept the reverse version
  + List.reverse() - doesn’t reverse the order alphabetically but instead reverses the index of the elements
  + len(List) - returns the number of elements in the list
  + Index error - an error if you ask for an index of the list but that index isn’t there

# 2. Variables and Simple Data Types

## Variables

* Variables - labels that hold a certain value
  + Can contain only letters, numbers, and underscore
    - Can’t start with a number
  + Spaces aren’t allowed
  + Avoid using python keywords or functions as variables
  + Variable names should be short but descriptive
  + Correct spelling of variables doesn’t matter
* Method - an action python can perform on a piece of data. variable.method()
* Traceback - a record of where the interpreter ran into trouble when trying to execute your code.
* Syntax Error - when python doesn’t recognize a section of the program
* String - series of characters that are between quotes. “Hello World”

| .title() | - changes the string so only the first letters of each word are capitalized |
| --- | --- |
| .upper() | - changes all the letters upper case |
| .lower() | - changes all the letters lower case |
| F-strings | - formats the string, placing any variable in brackets all between 2 quotes. You can also use the f-string in a variable |
| .rstrip() | - gets rid of any spaces at the right end |
| .lstrip() | - gets rid of any spaces at the left end |
| .strip() | - gets rid of whitespaces at both ends |
| \t | - tab |
| \n | - newline |
| \” | - double quote |

## Numbers

* Integers - number variables that can add, subtract, divide, and multiply
* Foat - any number with a decimal point
* When writing long numbers you can group digits using underscores
* You can initialize multiple variables using commas between names and the variables
* Constant - a variable that stays the same throughout the entire program. If you want a constant you have to have the name in all upper case
* Comments - allow you to write notes in the program. # indicates a comment
* Zen of python - “rules” that describe how to build a good python program

# How to access files

* + In the command prompt
    - Type: cd OneDrive\Desktop\python\_works
    - dir - lists all of the files in the python\_works folder
    - Then type: python file.py