Programming Notes

Jude Thaddeau Data

Documentation:
https://docs.python.org/3/tutorial/index.html

Table Of Contents:

SECTION 1- PYTHON OBJECTS & DATA STRUCTURES

Operations:
- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulo (%) is the remainder after division
- Exponents (**)
- Parenthesis ( ) are brackets that can be used to control order of operations

Number Methods:
- Hexadecimal: hex(integer)
- Binary: bin(integer)
- Exponential: pow(base, power, mod_exponent)
  - DEFAULT: mode_exponent = None
- Absolute Value: abs(number)
- Round: round(double, decmial_points)
  - DEFAULT: decimal_points = 0

Comments:
- 'Comments' are hashes (#) that when written is ignored by the program
- Can be used to provide notes when reading through code

Variable Assignments
- Data types can be assigned values like integers, floats, characters, strings, etc
- Example:

my_name = 'Jude Data'
- Do NOT start with a number & NO special symbols (e.g: :'",<>/? etc…)
- NO spaces are used; use underscore (_)

Writing & Formatting Code - PEP8
- Key words like 'list', 'str', etc cannot be used as variable names as they are 'special words'
- 'Special Words' are highlighted in different colours to note non-usability
- Use UPPER CASE words for global variables
- PEP8 DOCUMENTATION: https://www.python.org/dev/peps/pep-0008/

Python's Variable Assignment Flexibility:
- 'Dynamic Typing' means that variables can be reassigned different values of different data types (e.g: int to str, str to list, etc)
- Example:

my_name = 'Jude Data'
my_name = ['Jude', 'Data']
- Use type([insert variable name here]) function discover variable data type

STRINGS:
- Ordered sequences of characters
- Sections can be grabbed by 'indexing' or ‘slicing’
- [Square Brackets] are used after a string & every index starts at 0
- Positive Index: traversing forwards (e.g: 1, 2, 3, etc)
- Negative Index: traversing backwards (e.g: -1, -2, -3, etc)
- Can use 'SINGLE' or "DOUBLE" quotes

String Indexing Versus Slicing:
- 'Indexing' grabs a single character
- 'Slicing' grabs more than one character

Slicing Strings:
- Formula: [START:STOP:STEP]
- 'STOP' means up to BUT NOT INCLUDING
- 'STEP' refers to jumping size of index

Function - print()
- Used to print at least one line of characters
- 'Escape Sequences' can modify outputs (MORE online)
  - \t or 'tab'
  - \n or 'newline'
  - \b or ‘backspace'

Function - len()
- Returns the 'length' of a string
- Example: len("Jude Data") [output = 9]

String Indexing & Slicing Examples:
- NOTE: mystring = “abcdefghijk"
- msystring[0] = ‘a'
- mystring[10] = mystring[-1] = ‘k'
- mystring[3:6] = ‘def'
- mystring[2:] = ‘cdefghijk'
- mystring[::2] = ‘acegik'
- mystring[2:7:2] = ‘ceg'
- mystring[::-1] = ‘kjihgfedcba’

String Properties:
- 'Immutability' means that single elements within a data type CANNOT be changed after be assigned a value
- 'Concatenation' is the process of joining character strings together; a work around string immutability

Concatenation:

- Can be added together using single or multiple characters
- Can be multiplied by a factor to repeat the string multiple times
- NOTE: numbers of a character data type do NOT obey mathematical operations and obey string properties

Built In String Methods:

- 'Methods' are special functions added after a variable name with .methodname
- Examples: .upper() [ALL UPPER CASE], .split('character to be split on') [turns string into list], .format() [manipulates string layout], etc

String Formatting:

- Can be formatted using 'concatenation' or ‘formatting'
- String formatting involves the .format() method or f-strings (formatted string literals)

METHODS 1 - .format()

- SYNTAX: "String with {value:width.precision} “.format(value)
  - [A1] Index positions can be inserted into braces to manipulate the order of values
  - [A2] Variable assignments can be allocated to values then be inserted into braces
- NOTES:
  - #values = #curly_braces
  - include letter 'f' after 'precision' in curly braces for 'float formatting’
  - width = amount of white space used
  - precision = number of decimals used
- ALIGNMENTS:
  - < , ^ , > are left, centre, and right alignments
  - included BEFORE 'width' is stated
  - any character can be added before the alignment statement to substitute whitespace
- NOTE: 'width' & 'precision' are optional parameters

METHOD 2 - f-strings (Formatted String Literals)

- SYNTAX: f"String with {value:{width}.{precision}}”
- REQUIREMENT: If 'value' is a variable, variable MUST be ASSIGNED/CREATED before creating the f-string
- If 'float formatting' is needed, DROP the braces for width and precision & add letter 'f' after precision
- NOTE: 'width' & 'precision' are optional parameters for both methods

Advanced String Methods:
- Changing Case:
  - string.capitize() [converts first letter to capital]
  - string.upper()
  - string.lower()
- Location & Counting:
  - string.count('character(s)')
  - string.find('character(s)') [returns index]
- Formatting Methods:
  - string.center(string_length, 'character(s)')
  - string.expandtabs()
- Check Methods:
  - string.alnum() [checks if alphanumeric]
  - string.isalpha()
  - string.islower()
  - string.isspace()
  - string.endswith('character(s)')
  - string.istitle() [checks if all words start with upper case]
- Regular Expressions:
  - string.split(split_char)
  - string.partition(partition_char) [still includes character where partition occurs]
  - NOTE: split_char & partition_char are DEFAULT whitespace (' ')

LISTS:
- Are ordered sequences holding capable of holding different object types
- Objects are accessed via 'index position' like strings starting at 0
- Example: ["Jude", 20, "J117", 166.5]
- Supports 'indexing', 'slicing', 'nesting' & other methods

List Indexing & Slicing:
- Items within list are 'mutable' (can be reassigned/changed in value)
- Items can be concatenated using indexing or slicing

List Methods (More Online):
- .append() has the value within the brackets ADDED as the list's latest item [P]
- .pop() DELETES item on depending on index position supplied (default last index) [NP]
- sort() SORTS values depending on data type [P - DO NOT ASSIGN]
- reverse() REVERSES the order depending on data type
- NOTE:
  - [P] = Permanent
  - [NP] = Not Permanent (requires variable assignment)

Nested Lists:
- 'Nested Lists' are lists within lists
- Can have as many lists within lists provided computer can handle assignment
- Example:

Formula_1 = [["Ferrari", ["SV", "CL"]], ["Mclaren", ["LN", "CS"]], ...]

List Comprehension:
- 'List Comprehension' involves creating lists using 'for loops' and existing variables of appropriate data types
- Example:

```
F1_Drivers = [Team[1] for Team in Formula_1]
F1_Drivers
OUTPUT: [["SV", "CL"], ["LN", "CS"], ...]
```

- NOTE: list comprehensions can have nested 'for loops'

Advanced List Methods:
- Append: my_list.append(item)
- Count: my_list.count(item)
- Extending Lists: my_list.extend(another_list)
- Find Index: my_list.index(search_item)
- Insert: my_list.insert(index, inserted_item)
- Delete Item: my_list.pop(index)
- Delete 1st Occurance: my_list.remove(search_item)
- Reverse Permanently: my_list.reverse()
- Sort Permanently: my_list.sort(reverse)
  - DEFUALT: reverse = False

DICTIONARIES:
- Are unordered structures for storing objects via 'key-value' pairs
- Access the 'value' of an object by referring to its 'key'
- SYNTAX: { 'key1':'value1', 'key2': 'value2', ... }
- 'Value' can be of any data type such as a string, int, float, dict, list, etc
- Access dictionaries like lists but insert key name instead
- NOT immutable
- Lists Vs Dictionaries:
  - Lists: sorting, indexing or slicing is required
  - Dictionary: object's are name or value only needs to be called easy

Dictionary Methods:
- Dictionary Comprehensions: {key:operation for key in iterable}
- Methods for viewing or extracting keys, values & items:
  - my_dict.keys()
  - my_dict.values()
  - my_dict.items()

TUPLES:
- Similar to lists BUT uses parenthesis & objects inside tuples are 'immutable'
- Use when immutability is needed for data integrity
- Example: (1, 2, 3, Jude, Lilith)
- Tuple Methods (More Online):
  - .count() returns the number of times the value inserted appears
  - .index() returns the the index of the inserted value

SETS:
- Are unordered collections of UNIQUE elements
- Example: { 1 , 2, 7, ‘Jude'}

Creating Sets:
- METHOD 1: Variable Assignment

```
my_set = set()
my_set.add(1)
my_set
```

OUTPUT = { 1 }
NOTE: the function set() only take 1 argument
- METHOD 2: Creation

```
my_set = { 'Jude', 'Miguel', 'Miguel', 'Jude'}
myset
```

OUTPUT: {'Jude', 'Miguel'}

Set Methods:
- Adding: s1.add(item)
- Clearing: s1.clear()
- Copy: s1.copy()
- Difference: s1.difference(s2, s3, …)
  - NOTE: Returns the unique items within s1
- Assigning Difference: s1.difference_update(s2, s2, …)
- Discard: s1.discard(item)
- Intersection: s1.intersection(s2, s3, …)
  - NOTE: Returns the common elements between sets
- Assigning Intersection: s1.intersetion_update(s2, s3, …)
- Disjoint: s1.disjoiny(s2)
  - NOTE: Checks if there is NO intersection
- Subset: s1.issubset(s2)
- Superset: s2.issuperset(s1)
- Symmetric Difference: s1.symmetric_difference(s2)
- Union: s1.union(s2)
- Assigning Union: s1.update(s2)

BOOLEANS:
- Are operators that convey True or False statements
- Used in conjunction with comparison operators to help make conditions in statements
- FUNCTION: bool(condition) [returns whether a condition is True or False]
- Example:

```
type(FALSE)
```

OUTPUT = bool

GENERIC FILES:
- 'I/O File' means input-output file
- Used with text, audio, emails, excel files, etc (extra libraries may be needed)

Creating Files:
- Jupyter Notebook: %%writefile filename.txt
- Other: write file in VSCode or any other platform & SAVE with .txt extension

Opening Files:
- FUNCTION: open('filename.txt' [,access_mode][,buffering])
- NOTE: Check extra parameter types online
  - mode = 'r' (read only)
  - mode = 'w' (write only - overwrite)
  - mode = 'a' (append only)
  - mode = 'r+' (read & write)
  - mode = 'w+' (write & read - overwrite)
- BEWARE: the file to be opened MUST be in the same directory you are in OR provide FULL 'file path'
- To check directory: pwd (Terminal on MacOS)
- Windows:

```
myfile = open("C:\\Users\\YourUserName\\Home\\Folder\\myfile.txt")
```

- MacOS & Linux:

```
open("/Users/YouUserName/Folder/myfile.txt")
```

Reading Files:
- FUNCTION: .read(size)
- Returns the file contents up to 'size', default is entire file
- NOTE: .read() acts like a cursor going to end of file (EOF) [using default function twice results empty]
- Cursor reset FUNCTION: .seek(offset [,whence])
- By lines in list FUNCTION: .readlines(limit)
- NOTE: Check extra parameters online

Writing Files:
- Example:

```
myfile = open('filename.txt', 'w+')
myfile.write('This is a new line.')
myfile.seek(0)
myfile.read()
OUTPUT: 'This is a new line.'
myfile.close()
```

- NOTE: 'w' or 'w+' deletes all existing content in file [overwrite]

Appending Files:
- Intended to add additional content to files
- Example:

```
myfile = open('filename.txt', 'a')
myfile.write('\nThis is the 2nd line.')
myfile.seek(0)
print(myfile.read())
```

OUTPUT:
'This is a new line.'
'This is the 2nd line.'

```
myfile.close()
```

Closing Files:
- FUNCTION: .close()
- Opened files are NORMALLY held in memory thus they need to be closed before opening another
- ALTERNATIVE (new method with no file closing):

```
WITH open('filename.txt') AS variable_file_name:
    contents  = variable_file_name.read()
```

CSV FILES:
- 'CSV' means comma separated variables
- Common for spreadsheets
- ONLY stores raw data as an export (NO images, formulas, etc)
- Built-in csv module allows 'grabbing' columns, rows & values + 'writing' to customise a .csv file
- OTHER 3rd party libraries include pandas, openpyxl, etc
- CSV Documentation: https://docs.python.org/3/library/csv.html
- LIBRARY: import csv
- Example CSV content:

```
Name, Hours, Rate
David, 20, 15
Claire, 40, 20
```

Reading CSV Files:
- 'Encodings' (defualt: cp1252) is the process of putting a sequence of characters into a specific format for storage
  1. v1 = open('file_path', mode = 'r', encoding = 'utf-8')
     1. Modes are the SAME as normal files
  2. v2 = csv.reader(v1)
  3. v3 = list(v2)
  4. Variable 'v3' can now be manipulated to access content (e.g: len(v3))

Writing & Closing CSV Files:

- NOTE: opening files with mode 'w' overrides & deletes existing files with same name

1. v1 = open('file_path', 'w', newline = ' ')
    1. 'Newline' controls how universal newlines work (text mode only), e.g: ' ', '\n', '\r' & '\r\n'
2. v2 = csv.writer(v1, delimiter = ',')
    1. 'Delimiter' refers to the character that separates columns (can be ';', '\t', etc)
    2. NOTE: delimiter can ALSO be used in csv.reader(...) variable
3. v2.writerow(content_here) OR v2.writerows(contents_here)
4. v1.close()

PDF FILES:

- 'PDF' means portable document format
- Have no standard machine readable format; scanned PDFs are most likely unreadable OR have missing information (e.g: images, tables, etc)
- Fully function pdf modules are PAID; popular free software PyPDF2 (limited use)
- DOCUMENTATION: https://pythonhosted.org/PyPDF2/
- INSTALLATION:

>> python3 -m pip install PyPDF2

- LIBRARY: import PyPDF2

Reading & Adding To PDFs:

1. v1 = open('file_path', mode = 'rb')
    1. 'rb' means read binary
2. v2 = PyPDF2.PdfFileReader(v1)
3. Need to access contents
    1. #Pages: v2.numPages
    2. Specific Page: v3 = v2.getPage(index)
        1. Adding To PDFs:
            1. v4 = PyPDF2.PdfFileWriter()
            2. v4.addPage(v3) [Adding to this page]
            3. v5 = open('new_file', 'rb') [Contents here are to be added]
            4. v4.write(v5)
            5. v5.close()
    3. Page Text: v3 = v2.extractText()
4. Can now manipulate contents
5. v1.close()

SECTION 2 - PYTHON COMPARISON & LOGICAL OPERATORS

Introduction:
- Conditions can be implemented into 'control flow' & 'iteration' statements to direct order of code execution
- Conditions contain variables, numbers, data structures, etc that are compared & complimented by comparison & logical operators
- Comparison & logical operators are designed to impose a 'True' or 'False' boolean which ultimately directs control flow

Comparison Operators:
- [==] - checks if two operands are EQUAL
- [!=] - checks if two operands are NOT EQUAL
- [>] - checks if first operand is GREATER than latter operand
- [<] - checks if first operand is LESS than latter operand
- [>=] - checks if first operand is GREATER THAN OR EQUAL to latter operand
- [<=] - checks if first operand is LESS THAN OR EQUAL to latter operand

Logical Operators:
- 'AND' indicates that BOTH/ALL conditions must be satisfied to return TRUE
  - Example: (v1 <= v2) AND (v3 >= v4)
- 'OR' indicates that AT LEAST ONE condition must be satisfied to return TRUE
  - Example: (v1 == v2) OR (v3 != v4)
- 'NOT' will return TRUE if the condition is NOT satisfied
  - Example: (v1 < 117) AND (v2 NOT 44)

SECTION 3 - PYTHON STATEMENTS

Control Flow:
- 'Control Flow' executes a 'body' of code when a 'condition' is met
- Python uses colons & indentation (whitespace) to indicate control statements
- 'Conditions' often involve comparison or logical operators with variables or other data types like booleans, strings, etc
- Examples include keywords if, elif & else statements, for loops, while loops, etc

STATEMENTS - IF, ELIF, ELSE
- Allows computer to perform alternative actions based off a given set of results
- SYNTAX:

```
if (case_1):
    perform action 1
elif (case_2):
    perform action 2
else:
    perform action 3
```

- There can be BOTH ZERO OR MULTIPLE IF AND ELIF statements
- The else statement CAN BE omitted

ITERATION:
- 'Iteration' involves performing an action repeatedly until satisfied by a condition
- Elements like lists, sets, strings, tuples, dictionaries, etc are iterable
- Python allows iteration through FOR LOOPS & WHILE LOOPS
- Like other data types FOR & WHILE loops can be nested

FOR LOOPS:
- Allows computer to go through items in an ordered sequence
- SYNTAX:

```
for item in object:
    code statement(s)
```

- 'item' can be named anything & exists as a local variable within its domain
- 'object' can even be a function the range(start, stop)
- when the 'item' component is not to be mentioned at any point of the coding statement(s) an underscore '_' is appropriate

For Loop Techniques:
- Iterating through data types like tuples & lists can be cumbersome in syntax
- Also, lists containing tuples is a VERY COMMON data structure
- 'Tuple Unpacking' & 'Dictionary Unpacking' take advantage of this

Tuple Unpacking:
- Tuples are often elements within lists
- SHORTCUT SYNTAX:

```
for (t1,t2) in tuple_list:
    code statement(s)
```

- NOTE: parenthesis is option in syntax

Dictionary Unpacking:
- Normal for loops will refer to the key ONLY
- .keys() returns a list of all keys
- .values() returns a list of all values
- .items() returns a list of tuples containing all key:value items
- SHORTCUT SYNTAX:

```
for key,value in dict1.items():
    body of code
```

WHILE LOOPS:
- Executes a statement(s) while condition is TRUE
- SYNTAX:

```
while (condition):
    code statement(s)
```

- If 'condition' component is NOT carefully planned/updated this can lead to a non-terminating infinite loop

BREAK, CONTINUE & PASS:
- 'BREAK' breaks OUT of the current/closest enclosing loop
- 'CONTINUE' goes BACK to the top of the current/closest enclosing loop
- 'PASS' does nothing
- SYNTAX:

```
while (condition):
    code statement(s)
    if (condition):
        break
    if (condition):
        continue
```

Useful Functions Alongside Loops :
- Input:
  - Assigns keyboard input as a string by default
  - Can be enclosed by other functions like int() to convert to integer
  - SYNTAX: input(prompt)
- Range:
  - Used to generate a list of integers
  - SYNTAX: range(start, stop, step)
- Min & Max:
  - Returns the lowest or highest value of an object(s)/iterable respectively
  - SYNTAX 1: max(n1, n2, ...) or max(iterable)
  - SYNTAX 2: min(n1, n2, ...) or min(iterable)

USEFUL OPERATORS
- IN:
  - Used to check if an object is 'inside' another object
  - Example:
    'x' in ['x', 'y', 'z']
    OUTPUT: TRUE
- NOT:
  - Used to check if an object is NOT 'inside' another object
  - Example:
    'x' not in [1, 2, 3]
    OUTPUT: TRUE

LIBRARY:
- Are collections of data that when called integrate existing/written programs to the core program
- Prevents user from needing to make thing like functions from scratch
- IMPORT SYNTAX: import library_name
- SPECIFIC IMPORT SYNTAX: from library_name import function_name

Shuffle:
- Shuffles an iterable, changing it rather than returning a new one [DNA]
- SYNTAX: shuffle(iterable)
- Example:

```
from random import shuffle
mylist = [1, 2, 3]
shuffle(mylist)
```

Randint:
- Returns random integer including both endpoints
- SYNTAX: randint(start, stop)
- Example:

```
from random import randint
randint (0,100)
```

OTHER USEFUL FUNCTIONS:
Enumerate:
- Takes an iterable and matches the index followed by its respective item
- SYNTAX: enumerate(iterable, start)
- Example 1:

```
index = 0
for i,l in enumerate('ab')
    print(f"index: {i}, letter: {l}")
OUTPUT:
index: 0, letter: a
index: 1, letter: b
```

- Example 2:

```
list(enumerate(['apple', 'banana', 'cherry']))
OUTPUT: [(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

Zip:
- Pairs iterable items together based off index position in iterable
- SYNTAX: zip(iterable1, iterable2, …)
- Example 1:

```
l1 = [ 0, 1]
l2 = ['J',V']
l3 = ['D','S']
list(zip(l1, l2, l3))
OUTPUT: [(0, 'J', 'D'), (1, 'V', 'S')]
```

- Example 2:

```
for i1, i2 in zip(l2, l3)
    print(f"1st Item: {i1}, 2nd Item: {i2}")
OUTPUT:
1st Item: 'J', 2nd Item: 'D'
1st Item: 'V', 2nd Item: 'S'
```

EXTENSION - LIST COMPREHENSION
- Is an alternative method in building lists in a shorter notation
- NOTE: may be shorter in syntax but NOT computational time
- SYNTAX: [(operation) for (variable) in (iterable)

Example 1 - Standard

```
l1 = [letter for letter in 'Jude']
OUTPUT: ['J', 'u', 'd', 'e']
```

Example 2 - Operation Included

```
l2 = [x**2 for x in range(0,5)]
OUTPUT: [0, 1, 4, 9, 16]
```

Example 3 - If Statements

```
l3 = [x for x in range(5) if x%2==0]
OUTPUT: [0, 2, 4]
```

Example 4 - Complex Operation

```
l4 = [((9/5)*temp + 32) for temp [0, 10, 20.1])
OUTPUT: [32.0, 50.0, 68.18]
```

Example 5 - Nested List Comprehension

```
l5 = [x**2 for x in [x**2 for x in range(0,5)]]
OUTPUT: [0, 1, 16, 81, 256]
```

NOTE: There are MORE complex shorter notations BUT at the EXPENSE of READABILITY. Always PRIORITISE READABILITY.

SECTION 4 - METHODS & FUNCTIONS

Object Oriented Programming Terminology:
- 'Objects' is a collection of data (variables) & methods (functions) that act on data
- 'Method' is a function available to a specific object (lists: .append(), .pop(), etc)
- 'Class' is a template for creating objects
- 'Instance' is an individual object belonging to a class

Methods Introduction:
- Like functions, performs specific actions on an object
- SYNTAX: object.method(arg1, arg2, ...)
- Example (list methods): append, count, extend, insert , pop, remove, reverse, sort
- HELP FUNCTION:
  - Used to give information about a method
  - SYNTAX: help(object.method)
- More information: https://docs.python.org/3/

Functions Introduction:
- 'Functions' groups a set of statements so they can be used MULTIPLE times
- Can accommodate parameter(s)/argument(s) as inputs for manipulation
- USE: when a block of code needs to be used more than once
- Capable of being NESTED

Defining Functions:
- 'def' tells Python a function is being made
- 'Snake Casing' is a naming style whereby all words are lowercase & when needed, are separated by an underscore
- Parenthesis store input argument(s)/variable(s) to be manipulated
- Colon causes indentation to make statements belong to function
- 'Docstring' uses triple quotes to explain function as a comment
- 'return' (optional) keyword allows output of function to be assigned to a variable
- Can either 'return' or 'print'
- SYNTAX:

```
def name_of_function(arg1, arg2, ...)
    '''
    Docstring here
    '''
    (code statements here)
    ('return' or 'print' statement here [optional])
```

Calling Functions:
- Refer to function name with brackets
- Example: hello() [OUTPUT: hello]
- NOTE: calling function WITHOUT brackets makes python tell you it is a function

Input Parameters:
- Example (with default value):

```
def hello(name = 'Default'):
  print(f"Hello {name}")
```

- Call: hello('Jude') [OUTPUT: Hello Jude]

Return Versus Print:
- Return is used when the function output NEEDS to be SAVED via assignment
- Print ONLY displays output; NOT saves
- Assigning a non-return function to a variable results is 'NoneType'
- Example of return:

```
def add_num(n1, n2):
  return n1 + n2
add_num(1,2) [OUTPUT: 3]
```

- NOTE: functions can have BOTH return & print

Ambiguity Of Input Types:
- WARNING: ensure user inputs are of desired type
- Example

```
add_num('10','20') [OUTPUT: '1020']
```

Incorporating Logic Functions:
- Possible by incorporating boolean tests with return statement
- Example 1 (Checking for even integer):

```
def even_check(int):
    return int%2 == 0
```

- Example 2 (Checking if list has even number):

```
def even_in_list(num_list):
    for num in num_list:
        if num%2 == 0:
            return true
    # False at the end as search needs to exhaust list
    return false
```

- Example 3 (Returns all even numbers in list):

```
def even_nums(num_list):
    even_nums = []
    for num in num_list:
        if num%2 == 0:
            even_nums.append(num)
    return even_nums
```

- NOTE: in examples 2 & 3, if a 'false' boolean is in the wrong place of the if, elif or else statement, function becomes incorrect (ORDER LOGIC CAREFULLY)

Functions & Tuple Unpacking:
- 'Tuple unpacking' can be used in functions to return tuples
- Example:

```
work_hours = [('Abby', 100), ('Billy', 400), ('Cassie', 800)]
def employee_check(work_hours)
    current max = 0
    employee_of_month = ' '
    for employee, hours in work_hours:
        if hours > current_max:
            current_max = hours
            employee_of_month = employee
        else:
            pass
    return (employee_of_month, current_max)
```

- Returned tuples can have variables be assigned to them in a t1, t2 format
- Example:

```
name, hours = employee_check(work_hours)
```

Interactions Between Functions:
- The outputs of functions can be the inputs of another function
- Example:

```
# INITIAL LIST
mylist = [' ', 'O', ' ']
# SHUFFLE LIST
mixedup_list = shuffle_list(mylist)
# USER GUESS
guess = player_guess()
# CHECK GUESS
check_guess(mixed_list, guess)
```

- NOTE: mixedup_list is redundant as a simple ' shuffle_list(mylist) ' call already shuffles & CHANGES original

*ARGS & **KWARGS:
- Used as function arguments that can take in the same argument more than once
- *args takes an arbitrary number of 'arguments' as TUPLE values
- Example:

```
def myfunc(*args):
    print(args)
myfunc(1, 2, 3, 4) [OUTPUT: (1, 2, 3, 4)]
```

- **kwargs takes an arbitrary number of 'keyword arguments' with the function taking them as DICTIONARY inputs
- NOTE: Keyword CANNOT be expression
- Example:

```
def myfunc(**kwargs):
    print(kwargs)
myfunc(Jude = 20, Alexa = 26) [OUTPUT: {'Jude': 20, 'Alexa': 26}]
```

- args and kwargs can be combined together BUT ORDER MATTERS
- NOTE: 'args' and 'kwargs' are arbitrary names

MAP FUNCTION:
- Executes a specified function FOR EACH item in an iterable
- SYNTAX: map(function, iterables)
- Example:

```
def square(num): return num**2
nums = [1, 2, 3]
list(map(square, nums)) (OUTPUT: [1, 4, 9])
```

FILTER FUNCTION:
- Executes a boolean-output function FOR EACH item in an iterable & ONLY retaining items that pass a condition
- SYNTAX: filter(function, iterables)
- Example:

```
def check_even(num): return num%2 == 0
nums = [0, 1, 2, 3]
list(filter(check_even, nums)) (OUTPUT: [0, 2])
```

[NOTE: print 'map' or 'filter' as a list for readability]

LAMBDA EXPRESSIONS:
- For single-use functions to be implemented alongside 'map' & 'filter'
- SYNTAX: lambda input_name(s): (operation(s))
- Example 1:

```
list(map(lambda num: num ** 2, my_nums))
```

- Example 2:

```
list(filter(lambda n: n % 2 == 0,nums))
```

- Example 3 (multi-input):

```
lambda x,y: x*y
```

NESTED STATEMENTS & SCOPE:
- 'Scope' determines the usability of a variable to other parts of code via rules:

1) Name assignments will create or change local names by default
2) Program uses 'LEGB' rule to dictate order of variable use
- 'Local': names assigned within a function (def or lambda)
- 'Enclosing Function Locals': names declared in any enclosed/nested function [inner to outer]
- 'Global' (module): names assigned at top-level of a module file or assigned global in a def within the file
- 'Built-In' (Python): special names like 'open', 'range', etc

3) Global names SHARING input-variable names in functions are used UNTIL a variable of the same name is declared later

SCOPE DIAGRAM:

```
# Global
a = 1
def func1(a):
    # Local
    a = 2
    def func2():
        # Enclosing Function Local
        a = 3
```

Global Statement:

- Used to assign a value to a variable name from a local level, to global level
- SYNTAX: global variable name
- WARNING: Local operations on a global variable NOW EFFECT that variable OUTSIDE the local domain
- Example:

```
a = 1
def func():
    global a
    # variable 'a' has VALUE 1 from this point
    a = 2
    # variable 'a' has VALUE 2 from this point
```

- NOTE: It is NOT RECOMMENDED to use the values of variables declared outside a function especially if it does NOT share the same name as one of the local variables

Global Statement ALTERNATIVE:

- Ensure a function takes the global variable as an input and RETURNS the modified version of that variable
- Example:

```
global_var_name = func(global_var_name)
```

Extra Note:

- globals() & locals() functions can be used to check the current local and global variables within a program

SECTION 5 - OBJECT ORIENTED PROGRAMMING:

Terminology:
- Allows programmers to make custom objects that have methods & attributes
- 'Objects' involve data (variables) & methods (functions) that act on data
- 'Methods' are operations performed on specific objects (e.g (lists): .append(), .pop(), etc)
- 'Class' are blueprints for making future objects
- 'Instances' are objects made from a specific class
- 'Attributes' are characteristics of an object

OOP Syntax Summary:
- Names of classes contain follow 'Camel Casing' (Every word is CAPITALISED)
- 'class' declares a template for an object
- Statements with 'def' are declarations of methods
- __init__() is used to initialise attributes
- 'self' keyword links an attribute/parameter to the object/class
- 'Class Object Attributes' are attributes that are STATIC
- NOTE: methods can EITHER 'print' OR 'return'
- DIAGRAM:

```
class NameOfClass():

    # Class Object Attribute
    static_attribute = something

    def __init__(self, param1, param2, ...):
        self.param1 = param1
        self.param2 = (something, e.g: param1 + static_attribute)

    def some_method(self, param1, param2, ...):
        # perform some action
```

Class Keyword:
- Defines the details of a instantiated object
- SYNTAX: class NameOfClass():
- Each word should be CAPITALISED
- Example:

```
class Sample():
my_sample = Sample() ['my_sample' is an 'instance' of class 'Sample']
```

Creating Attributes:
- Are characteristics of objects
- SYNTAX:
  def __init__(self, attribute1, ...)
    self.attribute = something [e.g: string, boolean, etc]
- NOTE: the 'attribute' in 'self.attribute' DICTATES the variable name when called
- 'def' when used in classes is a declaration of a method
- '__init__()' executes code below it upon instantiation
- 'self' represents the instance of the object itself
- Example:

```
class Dog():
  # Attribute below
  def __init__(self, my_breed):
     self.breed = my_breed
my_dog = Dog(breed = ‘Lab')
```

- NOTE: when INSTANTIATING an class, attributes MUST be declared during this process too

Class Attributes:
- Refers to ANY attribute that are the SAME/STATIC for ANY instance of a class
- CALL SYNTAX 1: self.something = ClassName.CA_Name
- CALL SYNTAX 2: self.something = self.CA_Name
- Example:

[Every dog may have a DIFFERENT 'name', BUT ALL DOGS ARE ‘mammals']

```
class Dog():
     species = 'mammal'
     def __init()__(self, breed, name,...):
          self.breed = breed
          self.name = name
```

- NOTE: when calling 'attributes' OR 'class object attributes' NO PARENTHESIS is needed
- CALL SYNTAX: instance_name.attribute_name
- Example:

my_dog.species [OUTPUT: 'mammal']
my_dog.breed [OUTPUT: 'Shiba Inu']

OOP Methods:
- Are functions defined inside body of class & perform operations that MAY/ MAY NOT use an object's attributes
- NOTE 1: not every method needs parameters
- SYNTAX:

```
def method_name(self, user_param1, user_param2, ...):
    (body of operations)
    (return/print/nothing here)
```

- Example 1:

```
class Circle():
  pi = 3.14
  # Circle gets instantiated with a radius (default is 1)
  def __init__(self, radius=1):
      self.radius = radius
      self.area = radius * radius * Circle.pi

  # Method for resetting Radius
  def setRadius(self, new_radius):
      self.radius = new_radius
      self.area = new_radius * new_radius * self.pi
```

- CALL SYNTAX: instance_name.method_name(user_param1, …)
- Example 2:

```
c = Circle()
c.radius
c.area
c.setRadius(2)
```

Formatting Alongside Methods:
- When returning/printing class variables, self.attribute_name is the CORRECT syntax for .format() or f-strings
- For returning/printing non-class object names, a standard syntax is valid
- SYNTAX:

```
# Some class method:
def syntax(self, non_class_var):
    print(f"To print class variables we: {self.attribute_name}")
    print(f"For non-class variables we: {non_class_var}")
```

TYPES OF METHODS & REQUIRED DECORATORS:
- 'Static Method' can call without an instance, BUT CANNOT access other class methods or attributes [decorator: @staticmethod]
- 'Class Method' can call without an instance, can access class methods & attributes (BUT NOT instance [UNIQUE] attributes) [decorator: @classmethod, keyword: cls]
- 'Instance Method' can access everything BUT REQUIRES an instance [keyword: self]

Inheritance:
- Involves creating classes from OTHER classes that have already been defined
- 'Derived Classes' (descendants) have inherited a previously defined class
- 'Base Classes' (ancestors) are classes to be inherited
- NOTE: descendants OVERRIDE/EXTEND the functionality of ancestors (e.g: derived & base classes sharing a method of same name will PREFER derived)
- SYNTAX:

```
class BaseClassName():
    def __init__(self, param1, ...):
        (body of attributes/operations)

class DerivedClassName(BaseClassName):
    def __init__(self, param1, ...):
        BaseClassName.__init__(self):
        (body of attributes/operations)
```

Polymorphism Example:

```
class Animal():
    def __init__(self):
        print("Animal created.")
    def whoAmI(self):
        print("Animal")
    def eat(self):
        print(“Eating")

class Dog1(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog1 created.")
class Dog2(Animal):
    def __init__(self):
        Animal.__init__(self)
        Print("Dog2 created.")
    def whoAmI(self):
        print("Dog2")
    def bark(self)
        print("Woof!")
```

a = Dog1() [OUTPUT: Animal created.\nDog1 created]
b = Dog2() [OUTPUT: Animal created.\nDog2 created]
a.whoAmI() [OUTPUT: Dog1]
b.whoAmI() [OUTPUT: Dog2]

Polymorphism:
- Different classes MAY SHARE a method of the same name but python DOES NOT GET CONFUSED
- Example:

```
class Dog():
    def __init__(self, name):
        self.name = name
    def speak(self)
        return self.name+' says Woof!”

class Cat():
    def __init__(self, name):
        self.name = name
    def speak(self)
        return self.name+' says Meow!”

niko = Dog('Niko')
felix = Cat('Felix')
print(niko.speak()) [OUTPUT: Niko says Woof!]
print(felix.speak()) [OUTPUT: Felix says Meow!]
```

- Can be used with loops & functions:

```
for pet in [niko, felix]:
    print(pet.speak())
def pet_speak(pet):
    print(pet.speak())
```

Polymorphism & Abstract Classes:
- 'Abstract Class' are classes never expected to be instantiated
- 'raise' an 'exception handling' keyword to handle undesirable input
- Example:

```
class Animal()
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method.”)

class Dog(Animal):
    def speak(self):
        return self.name+' says Woof’

class Cat(Animal):
    def speak(self):
        return self.name+' says Meow!’

jack = Dog('Jack')
Jose = Cat('Jose')
print(jack.speak()) [OUTPUT: Jack says Woof!]
print(jose.speak()) [OUTPUT: Jose says Meow!]
```

- Other Real Life Examples:
  - Opening different file types (word, pdf, excel, etc)
  - Adding different objects

SPECIAL (MAGIC/DUNDER) METHODS:
- Allows the use of built in operations (e.g: len(), print(),etc) with user created objects DIRECTLY (i.e: SHORTCUT)
- NOTE: MORE SPECIAL METHODS ONLINE!
- Example:

```
class Book():
    def __init__(self, title, author, pages):
        print("Book created.")
        self.title = title
        self.author = author
        self.pages = pages
    def __str__(self):
        return f"Title: {self.title}, author: {self.author}, pages: {self.pages}"
    def__len__(self):
        return self.pages
    def __del__(self):
        print("Book destroyed.")

book = Book("Python", "Jude Data", 117)
print(book) [from __str__]
len(book) [from __len__]
del book [from __del__]
```

SECTION 6 - PYTHON MODULES & PACKAGES

PyPI Packages:
- 'PyPI' are collections for open-source third part python packages
- 'Standard Library' are in-built library available after downloading python
- PyPI Source: https://pypi.org/
- There are OTHER libraries/modules/packages used for specific tasks LIKE Django or Flask for web development
- A Google search looking for a package aimed at a task with python will lead to DOCUMENTATION pertaining to details like installation, upgrade, etc
- DO NOT USE: 'python' or 'pip' in command line or 'Terminal'
- USE: 'python3' & 'pip3' for PyPI packages

Downloading External Packages:
>> python3 -m pip install package_name
>> pip3 install package_name

Other Useful Commands On Terminal/Command Line:
>> python3 --version
>> pip3 --version
Updating PyPI
>> python3 -m pip install --upgrade pip setuptools wheel

Modules Versus Packages:
- 'Modules' are .py scripts containing code to be called & used by another .py script
- 'Packages' are a collection of modules
- 'Library' are a collection of modules or packages (e.g: Standard Python Library)

Custom Modules:
1) Create & save a .py script containing function(s)
2) CALL SYNTAX:
from module_name import function_name

Custom Packages:
1) Create a folder
2) For each folder (AND sub-folder), include an EMPTY script named __init__.py
3) Organise modules within folder (OR subfolder)
4) MAIN-FOLDER SYNTAX:
from package_name import module_name (as name)
5) SUB-PACKAGE SYNTAX:
from main_package_name.sub_package_name import module_name (as name)
6) FUNCTION CALL SYNTAX:
module_name.function_name()

__name__ AND __main__ :

- Used to check whether a code, functions, etc within a script/module is being RUN directly or being imported by something else (code understanding & organisation)
- 'Python' has ALL code at indentation lvl 0 gets executed, while in languages like 'C' code statements within the main() function ONLY get run
- 'Source Programs' or scripts that are RUN DIRECTLY to the command line will have a built in variable called "__name__" being assigned as "__main__"
- 'Non Source Programs' or imported modules will have __name__ being assigned as the filename (without .py)
- CAN ALSO BE USED to ensure a script can dually serve as a module & an a main program
  - Example: A module is a library, BUT it has a script mode to run unit tests
- SYNTAX:

```
if __name__ == "__main__":
    (body of statements executing code ONLY BELONGING TO this script)
```

SECTION 7 - ERRORS & EXCEPTION HANDLING

Introduction:
- 'Errors' involve user input that is deemed inappropriate for the program to continue running (e.g: expecting INT but got STR)
- 'Exceptions' are designed to notify users of a SPECIFIC error & have a body of code below to rectify it
- NOTE: more exception keywords: (https://docs.python.org/3/library/exceptions.html)

Try, Except, Else & Finally:
- 'Try' contains ANY code that CAN INVOKE AND EXCEPTION
- 'Except' contains code identifying a SPECIFIC ERROR & then HANDLING that error
- 'Finally' EXECUTES REGARDLESS if an error is found or not (optional)
- SYNTAX:

```
try:
    (code here)
except EXCEPTION1:
    (code here)
except EXCEPTION2:
    (code here)
else:
    (execute this code IF NO EXCEPTION)
finally:
    (code here)
```

- NOTE: except statements without a specific error will execute for ALL errors

UNIT TESTING - PYLINT & UNITTEST
- Used when projects expand to multi-file scripts
- Test files ensures individual CHANGES still allows code to run as expected
- 'Pylint' is a library that looks at code & reports possible errors (pep8, invalid, etc)
- 'Unittest' is a built-in library allows program testing to compare expected against returned results

Pylint:
- INSTALLATION:

```
>> python3 -m pip install pylint
```

- Create a file containing normal-executable code then run the following commands in command line:
  - SYNTAX 1: pylint file_name.py
  - SYNTAX 2 (statistics): pylint -ry file_name.py
- NOTE: 10/10 score often compromises machine readability over human

Unittest:
- Documentation: https://docs.python.org/3/library/unittest.html
- Examples of 'Test Types': TestFixture, TestCase, TestSuite, TestRunner, etc
- Examples of 'Asserts': assertTrue(), assertFalse, assertEqual, etc
- ASSERT SYNTAX: assertEqual(returned, expected)
- SYNTAX FORMULA:

```
import unittest
import module_name (as something)

class testing_class_name(unittest.TestType):
    def func_test_1(self):
        (code here)
       self.assertEqual(returned, expected)

    def func_test_2(self):
        (code here)
        self.assertTrue(boolean)
        self.assertFalse(boolean)

    def func_test_n(self):
        ...

if __name__ == "__main__":
    unittest.main()
```

SECTION 8 - PYTHON DECORATORS

Introduction:
- Used to tack extra features to an existing function (without changing it)
- Can be TURNED OFF by deleting the line with the '@' operator COMMENT
- Possible because functions can BOTH accept & return other functions
- NOTE: most of the time, function decorations will be imported (e.g: django, flask, etc) rather than made
- SYNTAX:

```
@decorator_name
def original_func():
    (code here)
```

Example Of Decorators:
- NOTE: pass the original function as an argument, do so WITHOUT parenthesis

```
def decorating_func(original_func):
    def extra_feature():
        (code here BEFORE executing original function)
        original_func()
        (code here AFTER executing original func)
    return extra_feature()

# Can now place decorator operator (@) above original function
@decorating_func
def original_func():
    (code here)
```

SECTION 9 - PYTHON GENERATORS

Introduction:
- Makes a sequence of values overtime (rather than WASTEFULLY storing in memory)
- 'yield' keyword outputs value after value without storing past values
- Functions using 'yield' need to iterate or cast (e.g: list(), tuple()) to display output)
- 'State Suspension' is a characteristic of generators & means that when single values are computed, activity is suspended until next instruction
- Example:

```
def fib(n):
  a = b = 1
  for i in range(n):
      yield a
      a,b = b, a+b
```

Next Function:
- Takes a generator assigned instance & returns the next item in sequence
- 'defualt' (optional) is the returned value after iterable has reached END
- SYNTAX: next(iterable, default)

Iter Function:
- Takes an 'iteratively exhausted' object & makes it iterable again
- 'sentinel' is the value if matched, will stop iteration
- SYNTAX: iter(object, sentinel)

SECTION 10 - ADVANCED PYTHON MODULES

Advanced Module Contents:
- Collections
- OS module & Datetime
- Math & Random
- Python Debugger
- Timeit
- Regular Expressions
- Unzipping & Zipping Modules

COLLECTIONS MODULE:
- Built-in module that implements specialised container data types
- SYNTAX: import collections

counter:
- A 'dict' subclass that counts hashable objects
- Takes iterables & returns a dictionary containing keys of single elements & values of their frequency
- IMPORT SYNTAX: from collections import Counter
- SYNTAX: Counter(iterable)
- Common operations (given: c = Counter(iterable)):
  - c.most_common(n) [list 'n' most common key,values as tuples]
  - sum(c.values()) [total of all counts]
  - c.clear() [reset all counts]
  - list(c), set(c), dict(c), c.items() [self explanatory]
  - Counter(dict(list_of_pairs)) [convert from a list of (element,counter) pairs]
  - c.most_common()[:-n-1:-1] ['n' least common elements]
  - c += Counter() [remove zero & negative counts]

defaultdict:
- Assigned to existing dictionaries & returns a default value for non-existing keys
- IMPORT SYNTAX: from collections import defaultdict
- SYNTAX: my_dict = defaultdict(defualt_val)
- Example:

```
my_dict = {'FirstName' : 'Jude', 'LastName': 'Data'}
my_dict[‘Age’] = defaultdict(lambda: 0)
my_dict['Age'] [OUTPUT: 0]
```

namedtuple:
- Produces tuples where elements can be called by 'name' rather than ‘index'
- Quick method of creating creating a new object/class type with attributes
- IMPORT SYNTAX: from collections import namedtuple
- SYNTAX: Variable_Name = namedtuple(Class_Name, ['atr1', 'atr2', …])
- Example:

```
Dog = namedtuple('Dog', ['age', 'breed', 'name'])
Jack = Dog(age = 2, breed = 'Lab', name = 'Jackson')
Jack.age [OUTPUT: 2]
```

OPENING + READING FILES & MODULES (SHUTIL & OS MODULE):
- Allows files & directories to be navigated & have actions be performed on them (e.g: moving, deleting, etc) [BOTH built-in]
- OS Documentation: https://docs.python.org/3/library/os.html
- Shutil Documentation: https://docs.python.org/3/library/shutil.html

OS Syntax:
- LIBRARY: import os
- Getting directories: os.getcwd()
- Listing files in a directories: os.listdir(path) [default current]
- Walking through a directory: os.walk(file_path)
- Deleting files (2 WAYS):
  1) Deletes file at provided path: os.unlink(path)
  2) Deletes EMPTY folder at provided path: osrmdir(path)

SHUTIL Syntax:
- LIBRARY: import shutil
- Moving files (NOTE permissions): shutil.move(file_path/file_name, moved_path)
- Deleting files [DANGEROUS + PERMANENT]: shutil.rmtree(path)

Send2trash Syntax:
- INSTALLATION:
>> python3 -m pip install send2trash
- LIBRARY: import send2trash
- Moving files to bin: send2trash.send2trash(file_name)

DATETIME MOUDLE:
- Used to deal with timestamps
- SYNTAX: import datetime
- EXTRA: from datetime import datetime

Time:
- Only holds values of time (NOT DATE)
- SYNTAX: v = datetime.time(hour, minute, second, microsecond)
- Hours: v.hour
- Minutes: v.minute
- Seconds: v.second
- Microsecond: v.microsecond
- Time Zone: v.tzinfo

Arithmetic:
- Can be performed on 'date' or 'datetime' objects
- Example:
d1 = datetime(2021, 11, 3, 22, 0)
d2 = datetime(2020, 11, 3, 12, 0)
r = d1 - d2
r.total_seconds() [OUTPUT: 31572000.0]

Dates:
- SYNTAX 1: v = datetime.date(year, month, day)
- SYNTAX 2: v = datetime.datetime(year, month, day, hour, minute, second)
- SYNTAX 3: v = datetime.date.today()
- Day: v.day
- Month: v.month
- Year: v.year
- Everything: v.ctime()
- Tuple: v.timetuple()
- Ordinal: v.ordinal()
- Replace: v = v.replace(unit = value)

MATH MODULE:
- NOTE 1: these notes DO NOT COVER EVERYTHING IN MODULE
- NOTE 2: for more complex math operations, use NUMPY (from pypi)
- SYNTAX: import math
- Command Details: help(math)

Round:
- Round Up: math.ceil(value)
- Round Down: math.floor(value)
- Normal: round(value, places)

Constants:
- math.pi
- math.e
- math.tau
- math.inf
- math.nan

Logarithms:
- math.log(value, base)
- NOTE: base default ‘e'

Trigonometric Functions:
- NOTE: values default as radians
- math.sin(value)
- math.cos(value)
- math.tan(value)
- math.radians(value_in_degees)
- math.degrees(value_in_radians)

RANDOM MODULE:
- NOTE: these notes DO NOT COVER EVERYTHING IN MODULE
- SYNTAX: import random
- Command Details: help(random)

Seed:
- Used to initialise random number generator for a sequence of set values
- SYNTAX: random.seed(value, version)

Random Integer:
- Used to generate a random number (both endpoints inclusive)
- SYNTAX: random.randint(start, end)

Random With Sequences:
- With Replacement: random.choice(iterable, sample_number)
- Without Replacement: random.sample(iterable, sample_number)
- Shuffling (DO NOT ASSIGN): random.shuffle(iterable)

Random Distributions:
- NOTE 1: start & endpoints inclusive
- NOTE 2: both distributions below are continuous
- Uniform: random.uniform(start,end)
- Gauss: random.gauss(mu, sigma)

PYTHON DEBUGGER MODULE:
- Sets a trace to pause python execution mid-script to allow users to investigate code
- Allows variables, functions, etc to be executed mid-script
- Alternative to using 'print' statements
- SYNTAX: import pdb

Setting A Trace:
- Select a line to set trace from
- The program will execute up to the point of trace
- NOTE: still requires user to look at error code to solve problem(s)
- SYNTAX: pdb.set_trace()

REGULAR EXPRESSIONS MODULE:
- 'Regex' (RE) are used to search for a 'patterned structure' of a string
- E.g: phone number format; (012)-345-6789
- RE Example Pattern 1: r”(\d\d\d)-\d\d\d-\d\d\d\d"
- RE Example Pattern 2: r”(\d{3})-\d{3}-\d{4}"
- The 'r' in-front of the strings indicates a 'pattern format' (use of quantifiers)
- Documentation RE Library: https://docs.python.org/3/library/re.html
- Documentation Regex: https://docs.python.org/3/howto/regex.html
- LIBRARY: import re

Basic Patterns:
- EARLIEST MATCH: m = re.search(pattern, text)
  - Used Indexes: m.span()
  - FIRST Index: m.start()
  - LAST Index: m.end()
- ALL MATCHES: m = re.findall(pattern, text)
  - MATCHES COUNT: len(m)
  - SPECIFIC MATCHES: m.group(index) [NOTE: index starts at 1]
- COMBINING EARLIEST + ALL MATCHES:

```
for match in re.finditer(pattern, text):
    # Can use .span(), .start(), .end(), etc methods for every found 'match'
    (code here)
```

Regular Expressions Syntax:
- Regex SYNTAX: r”pattern_here"
- 'r' indicates that ANY '\' are NOT escape slashes
- '\' are used to indicate the use of specific type of 'character identifiers' or ‘quantifiers'
- Regex patterns are USED ALONGSIDE the re library

Character Identifiers:
- DIGITS: \n
- ALPHANUMERIC (includes underscore): \w
- WHITE SPACE: \s
- NON-DIGIT: \D
- NON-ALPHANUMERIC: \W
- NON-WHITESPACE: \S

Quantifiers:
- Occurs 1 or more times: +
- Occurs exactly 'n' times: {n}
- Occurs 'n' between 'm' times: {n,m}
- Occurs 'n' or more times: {n,}
- Occurs 0 or more times: *
- Occurs once or none: ?

Groups:
- Allows patterns to be dissected further with .group(index) function
- SYNTAX: pattern = re.compile(r"(s1)-(s2)...") [insert into re.search()]
- Parenthesis are used to break down a pattern into segments
- E.g: finding phone number + area code (1st 3 numbers)

Common Regex Operators:
Wildcard:
- '.' is used to return matches INCLUDING PRIOR elements
- E.g: re.search(r"...at", "the bat got splat.”)

Or:
- '|' is used for multiple patterning matching
- E.g: re.search(r"man|woman", "There were men & women.")

Starts or Ends With:
- Start ('^') e.g: re.search(r"^\d", "1 start.")
- End ('$') e.g: re.search(r"\d$", "End 2")

Exclusion:
- Any pattern inside '[^]' is excluded
- E.g: re.findall(r"[^\d]+", "3 in here.")

Brackets For Grouping:
- Can combine identifiers & qualifiers in brackets for patterns (e.g hyphenated words)
- E.g: re.findall(r"[\w]+-[\w]+", "hyphen-words are long-ish.")

Parenthesis For Multiple Options:
- Parenthesis is used for extended matching
- E.g: re.search('cat(nip|fish)', 'catnip or catfish?')

MODULE FOR TIMING PYTHON CODE:
- LIBRARY: import timeit
- All code should be time-tested for very large inputs
- SYNTAX: timit.timit(statement, setup, number)
  - NOTE: 'statement' & 'setup' are PASSED AS STRINGS (best using triple quotes)
  - 'statement' refers to the line(s) of code to be executed
  - 'setup' refers to the ENTIRE function definition
  - 'number' refers to the amount of times the 'statement' & 'setup' will be ran

ZIPPING & UNZIPPING FILES:
- 'Zip Files' are compressed files that take less space
- Need to be 'unzipped' when downloaded and opened
- LIBRARY: import zipfile

Creating Zip Files, Compressing & Extracting Files (INDIVIDUALLY):
1. z = zipfile.ZipFile(zip_name, mode = 'w') [CREATING ZIP FILE]
2. z.write(file_name, compress_type =zipfile.ZIP_DEFLATED) [ZIPPING FILES]
3. z.close()
4. zip_obj = zipfile.ZipFile(zip_name, mode = 'r')
5. zip_obj.extractall(extract_name) [EXTRACTING FILE]

SHUTIL Library For Handling Zip File Archives (WHOLISTIC):
- Creating Archive: shutil.makearchive(arhive_name, 'zip', path)
- Extracting Archive: shutil.unpack_archive(archive_name, extract_name, 'zip')

SECTION 11 - INTRODUCTION TO WEB SCRAPING

Introduction:
- Involves using techniques that automate data gathering from a website
- Data may include images, specific words, etc

Website Basics:
- 'Front-End' is the interface the user interacts with when a browser loads a website
  - Browser uses URLs to connect to a website
  - Websites passes code (e.g: HTML, CSS & Javascript) to browser which then displays contents
  - Web scraping programs grab specific portions of HTML code for data
- 'Back-End' involves the logic that provides functionality to a website (not visible to client/user)

HTML & CSS:
- 'Hypertext Markup Language' is what is used to display information on a website
  - HTML: https://developer.mozilla.org/en-US/docs/Web/HTML
- 'Cascading Style Sheets' formats & styles (colour, font, animation, etc) a website
  - CSS: https://developer.mozilla.org/en-US/docs/Web/CSS

Rules Of Web-scraping:
- Ask permission before scraping
- Overflow of scraping requests can lead potential IP address blocking

Limitations Of Web Scraping:
- Websites can change & current scraping code can become non-functional
- Period updates to scraping code may be required

Front-End Components Of A Website:
1. HTML (Formats & structures text like paragraphs, titles, etc)
2. CSS (Controls styles like colour, image size, etc)
3. Javascript (Reads, stores & manipulates data to be displayed by HTML & CSS)
   - Python can be used extract specific sections from the first 2 components
   - Browser does NOT access all source code, only HTML & some CSS + Javascript

HTML General Format:

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Title on Browser Tab</title>
    </head>
  <body>
    <h1> Website Header </h1>
    <p> Some Paragraph </p>
   <body>
</html>
```

HTML Notes (SEE DOCS FOR MORE TAGS):

- DOCTYPE tells browser that file is HTML
- 'Tags' are either 'opening' or 'closing' (denoted by '/')
- ALL component blocks are placed between <html> & </html>
- Meta data & script descriptions (CSS or JS files) are often placed in <head> block
- <title> denotes title of webpage
- Blocks in <body> & </body> is what is visible to user
- Headings range from <h1> to <h6>, number denotes heading size
- Paragraphs are stored within <p> tag which is simply text

CSS & HTML Example:

```
<!DOCTYPE html>
  <html>
      <head>
    <link rel = "stylesheet" href = "styles.css">
    <title>Some Title</title>
    </head>
    <body>
      <p id = 'para2'> Text </p> (OR) <p class = 'cool'> Text </p>
    <body>
</html>
```

CSS & HTML Notes:

- 'link' connects html components to certain file types
- 'rel' (relationship) indicates what type of file is being connected
- 'href' is a source reference to the relationship file
- 'Tags' (e.g: id, class, etc) define which html elements will be styled
- 'id' are used for single use & unique styles for a specific HTML tag
- 'class' define styles that can be linked to MULTIPLE html tags

CSS Example Code:

```
/* Code for ALL PARAGRAPHS */
p {
  color: red;
  font-family: courier;
  font-size: 160%;

/* Example ID */
.classname {
  color: red;
  font-family: verdana;
  font-size: 300%
}

/* Example CLASS */
#idname {
  color: blue
}
```

CSS Syntax:
- 'Hash' (#) declares an 'id'
- 'Period' (.) declares a 'class'

General Web Scraping Method:
- HTML contains information
- CSS implements styles
- Python can scan through HTML & CSS to locate specific information on a page
- Python uses 'request' LIBRARY to point to specific tags
- NOTE: HTML & CSS tags are differentiated by colour code

WEB SCRAPING LIBIRARIES:
- Command Line Installation:

```
>> python3 -m pip install requests
>> python3 -m pip install lxml
>> python3 -m pip install bs4
```

- Importing in Python:

```
import requests
import bs4
```

- Requests: https://requests.readthedocs.io/en/master/
- BS4: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Retrieving Source Code:
1. Specific Sections (right click component) -> inspect
2. ALL Information (right click blank space) -> view source page

Grabbing A Title:

1. Use requests library to grab page
    1. variable = requests.get("website_url")
        1. v1.text [View content in HTML]
2. Use bs4 (BeautifulSoup) library to organise & analyse page
    1. v2 = bs4.BeautifulSoup(v1.text, "string_code")
        1. v3 = v2.select('tag_name') [Grabs elements & returns list)
        2. v3[index].getText() [Returns the string version]

Grabbing A Title Example:

```
page = requests.get("http://en.wikipedia.org/wiki/Bruce_McLaren")
print(page.text)
organised_page = bs4.BeautifulSoup(page.text, 'lxml')
print(organised_page)
title_tag = organised_page.select('title')
print(title_tag[0].getText())
for item in title_tag:
    print(item.text)
```

Grabbing A Class:

1. 'Inspect' information of interest & find class assignment name
2. Assign request & BeautifulSoup variables to extract & analyse source
3. Implement class assignment name into select() method
    1. v.select('class_name')

CSS Class Syntax For .select() (LIMITED):

- v.select(‘div’)
  - All elements with tag
- v.select(‘#some_id’)
  - The HTML element containing the id attribute of some_id
- v.select(‘.notice’)
  - All the HTML elements with the CSS class name 'notice'
- v.select('div span’)
  - Any elements name that are within an element named
- v.select('div > span’)
  - Any elements named that are directly within an element named , with no other element in between

Grabbing Images:

- Images typically have their own URL link (e.g: .png, jpg) as a source
- bs4 can scan a page, locate & grab <img> tags + URLs
- URLs can then be downloaded as images & then 'write' them

Grabbing Images Example:

```
page = requests.get("http://en.wikipedia.org/wiki/Bruce_McLaren")
organised_page = bs4.BeautifulSoup(page.text, 'lxml')
image_info = organised_page.select('.thumbimage')
first_image = image_info[0]
# NOTE: types like bs4.element.tag can be used as dictionaries
first_image['src']
image_link = requests.get('image_url')
image_link.content [returns binary representation of image]
# Images can be written & saved
f = open('filename.jpg', 'wb') ['wb' = write binary]
f.write(image_link.content)
f.close()
```

SECTION 12 - INTRODUCTION TO PYTHON WITH EMAILS

SENDING EMAILS:

- Need to connect to email server, confirm connection, protocol setup, log into email, & send message
- LIBRARY: import smtplib
- 'SMTP' means 'simple mail transfer protocol'; a domain name used to access email via programs (e.g: GMAIL: smtp.gmail.com - need to generate app password)

Email Sending Steps:

1. smtp_object= smtplib.SMTP('server_domain_name', port-number = 587)
2. smtp_object.ehlo()
3. smtp_object.starttls() [NOT NEEDED IF port = 465]
4. https://support.google.com/accounts/answer/185833?hl=en/
    1. Sets up app password + 2-step verification
    2. Choose mail as app & name it + save app password
5. email = getpass.getpass('Email: ')
6. password = getpass.getpass('Password: ')
    1. import getpass
    2. SAVE PASSWORD ELSEWHERE
7. smtp_object.login(email,password)
8. from_address = getpass.getpass("User email: ")
9. to_address = getpass.getpass("Recipient email: ")
10. subject = input('Subject: ')
11. message = input('Message: ')
12. msg = "Subject" + subject + "\n" + message
13. smtp_object.sendmail(from_address, to_address, msg)
    1. NOTE: {} means success
14. smtp_object.quit()

TRAVERSING (RECIEVING) EMAILS:

- LIBRARY 1: import imaplib
- Has keywords:
    1. 'ALL' or
    2. 'BEFORE date', 'ON date', 'SINCE date'
        1. date format: dd-Mon-yyyy
    3. 'FROM some_string', 'TO some-string', 'CC some_string', 'BBC some_string', 'SUBJECT string', 'BODY string', 'TEXT string'
        1. string can be an 'email', 'subject name', etc
    4. 'SEEN', 'UNSEEN', 'ANSWERED', 'UNANSWERED', 'DELETED', 'UNDELETED', etc
- LIBRARY 2: import getpass
- LIBRARY 3: import email
    - Used to grab messages

Email Browsing Steps:

1. my_mail = imaplib.IMAP4_SSL('imap.gmail.com')
2. email = getpass.getpass("Email: ")
3. password = getpass.getpass("Password: ")
    1. USE app password
4. my_mail.login(email, password)
5. my_mail.list()
6. my_mail.select('inbox')
    1. Can now use imaplib syntax for specific mail
7. typ, data = my_mail.search(None, 'imaplib_keyword')
    1. if variable 'data' does NOT return any numbers: no matches
8. email_id = data[index]
9. result, email_data = my_mail.fetch(email_id, '(RFC822)')
    1. print(email_data) to see format
10. raw_email = email_data[index1][index2]... (indexes needed depends on email structure)
11. raw_email_string = raw_email.decode('utf-8')
12. email_message = email.message_from_string(raw_email_string)
13. Can now parse through email contents

SECTION 13 - INTRODUCTION TO PYTHON WITH IMAGES

Introduction:
- 'PIL' means 'Python Image Library'
- INSTALLATION:

>> python3 -m pip install pillow
- LIBRARY: from PIL import Image

Opening & Saving Images:
- v = Image.open('file_path')
- v.show()
- v.save('file_path')

Image Information:
- v.size [returns tuple: (width, height)]
- v.filename
- v.format_description

Cropping Images:
- Take cropped image 'v' & paste it on top of raw image 'v' within the frame (c1, c2)
- v1 = v1.crop((x,y,w,h))
- v2.paste(im = v1, box = (c1, c2), mask = None)

Resizing & Rotating Images:
- v = v.resize((c1, c2))
- v = v.rotate(degree, expand) [default 'False' for expand]

Image Transparency:
- Most images have RGBA (red, green, blue, alpha) system
- v.putalpha(integer) [integer ranges from 0-255]
- The higher the integer the LESS TRANSPARENT

SECTION 14 - DATA WRANGLING

TUTORIAL 1 - PANDAS

Pandas & IPython Library:

- 'Pandas' is a data science library mainly used for reading data
  - Includes data structures like 'DataFrames' & 'Series'

    >> import pandas as pd
- 'IPython' contains functions to neatly display series & data frames

    >> from IPython.display import display
- DOCS: https://pandas.pydata.org/docs/user_guide/index.html#user-guide
- Installation:

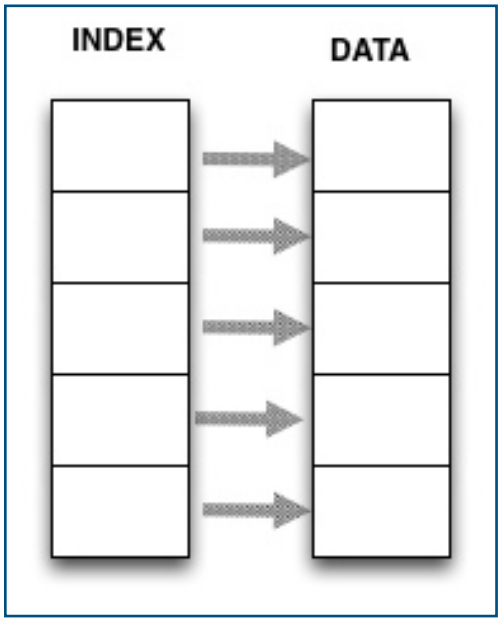
    >> python3 -m pip install pandas

    >> python3 -m pip install IPython

SERIES:

- 'Series' stores an array of indexes & data pointed by these indexes
- Uses index-value relationship (like a list)
- DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html
- METHOD:

    >> variable = Series(data, index=index)
- 'Data' contains information like strings, letters, etc
- 'Index' is used to access data & via numbers [lists] OR keys [dictionaries]



Example Of Series:

```
# Standard
>> data1 = [1, 2, 3, 4]
>> v1 = pd.Series(data1)

# With Specified Index
>> data2 = [4, 3, 2, 1]
>> index2 = range(100,105)
>> v2 = pd.Series(data2, index2)

# Displaying Data/Values
>> display(v1.values)

# Displaying Indexes/Keys
>> display(v2.index)

# Reassigning Indexes
>> v2.index = range(…)

# Using Dictionaries As Data & Index
>> data3 = {'Jude': 312, 'John':117, 'Jorge': 52, 'Carter': 259}
>> v3 = pd.Series(data3)
```

Slicing Series:
- 'Series' & 'Data Frames' can be 'sliced' like lists & strings
- Equivalent of SELECT * FROM columns WHERE condition for databases
- .loc[ ] locates matching values (inclusive)
  - SYNTAX: dataframe.loc[dataframe[col] (comparison operator) value]
  - DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html
- .iloc[ ] locates matching index values according to an index (avoid unless 'sub-sampling')
- Example:

```
# Get values (spartan service numbers) greater than 117
>> display(v3.loc[v3 > 117])
# Slicing for 312
>> v3['Jude']
```

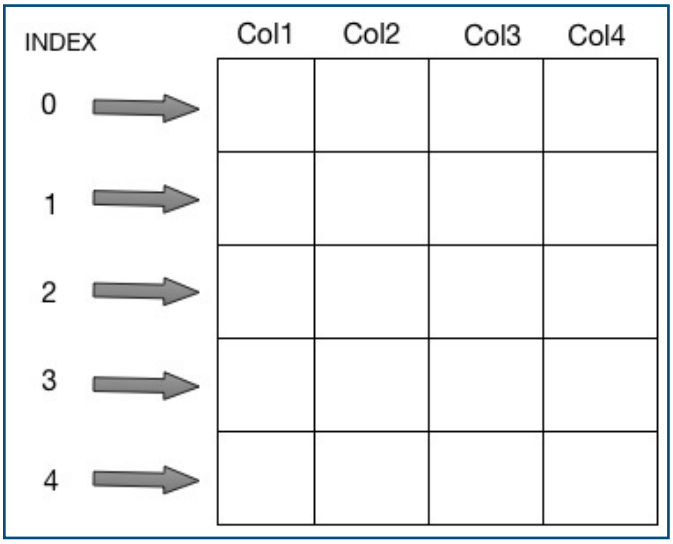
Series Operations:
- Operations can be applied according to data type
- Functions can be applied to columns
- Example:

```
>> double_v3 = v3 * 2
>> display(double_v3)
```

Series Methods:
- NOTE: more methods in documentation
- Can take the 'mean', 'standard deviation', 'cumulative sum', etc
- Redefine column or index name, etc
- Example:

```
>> v3.mean(), v3.std()
>> v3.name = 'Spartans'
>> v3.index.name = 'First Name'
# Gets the count, means, std, min &
max, & Q1 + Q2 + Q3
>> v3.describe()
```

DATAFRAMES:
- 'DataFrames' are a tabular at a structure that can store multiple rows & columns (i.e: like spreadsheets)
- Each 'row index'; corresponds to a 'row of column values'
- NAMING CONVENTION: df_something
- Indexed like dictionary keys: df[column_name]
- DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/frame.html



DataFrame Example:

```
>> population_data = {'1990':17065100, '2000':19153000, '2007':20827600,
'2008':21249200,'2009':21691700,'2010':22031750, '2011':22340024,
'2012':22728254, '2013':23117353}
>> population = pd.Series(population_data)
>> emission_data = {'1990':15.45288167, '2000':17.20060983,
'2007':17.86526004, '2008':18.16087566,'2009':18.20018196,'2010':16.92095367,
'2011':16.86260095, '2012':16.51938578, '2013':16.34730205}
>> emission = pd.Series(emission_data)
>> df = pd.DataFrame({"Emission": emission, "Population": population})
>> display(df)
# Get FIRST 5 (default) values
>> display(df.head())
# Get LAST 5 (default) values
>> display(df.tail())
# Get FIRST 10 values
>> display(df.head(10))
```

Reading CSV Files:

- METHOD: pd.read_csv(filename)
  - Optional arguments: encoding, astype, etc
- DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
- Example:

```
>> countries = pd.read_csv("data/countries.csv", encoding = "ISO-8859-1")
>> display(countries)
```

DataFrame Methods (NOT ALL):

- DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/frame.html
- df[column_name].value_counts()
  - Calculates the frequency of each data value
- df.groupby(column_name or column names as an iterable)
  - Groups results by a common element
- df.apply(function, *args, **kwargs)
  - Applies function to a data frame
- df.sort_values(by = column_name/column(s) as an iterable, ascending = True)
  - Sorts value along either axis (index = 0, columns = 1)
- df.map(dictionary specifying value to new value)
  - Substitutes each value in a series with another value (derived: func, dict, etc)
- df.update(new version of the data frame)
  - Modify series in place using values passed Series
- df.reset_index(drop = True for remove index or False for keeping index)
  - Resets the index of a DataFrame & use default one instead
- df.fillna(value)
  - Fills in segments with NaN with the 'value'
- df.rename({original_name: dersired_name}, axis = 1, inplace = True)
  - 'axis =1' indicates to change column names (axis = 0 for index)
  - 'inplace = True' specifies we want to 'mutate' the original

NaN Data Type:
- 'NaN' or not a number is equivalent to None & implies missing data
- Evaluates to 'True' & of type 'float'
- Comparisons should be used with np.NaN

```
>> import numpy as np
```

Utilising Subsets - Shallow Versus Copy
- 'Shallow' subset values are sliced sections that when changed, also changes the value of any variable utilising it
- 'Copy' keeps a copy of the initial assignment of a subset value
  - METHOD: .copy()
  - Example:

```
>> df_2010 = df2['2010'].copy()
>> df2['2010'] = df2['2010'].apply(lambda x: x+ 99 if x != np.NaN else -1)
# Remains unchanged despite reassignment of df2
>> display(df_2010)
```

Sorting Operations:

```
# Ascending Order
>> df_2010.sort_values()
# Descending Order
>> df_sort_values(ascending = False)
# Sorting Column Values Of A DataFrame
>> sorted_2012 = df2.sort_values(by='2012', ascending = False)
# Sorting Column Values Using Two Columns
>> sorted_2012 = df2.sort_values(by=['2012', '2013'], ascending=[False,True])
```

Slicing DataFrames Using .loc & .iloc Method Examples:

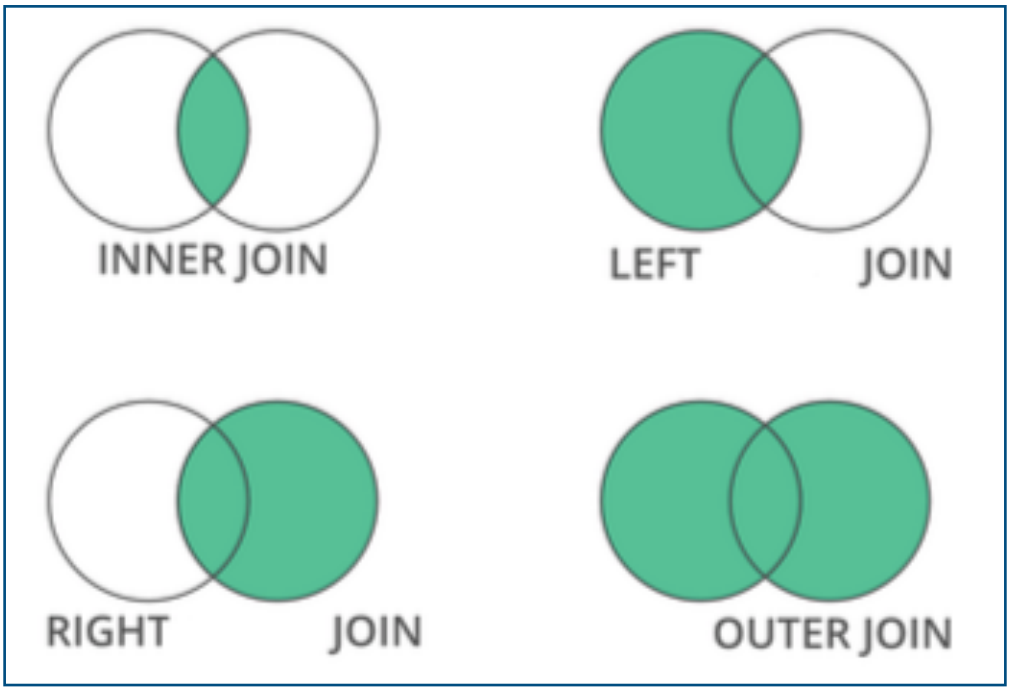
```
# Give me all the '2013' row values if the row in '2012' was greater than 40
>> display(df2.loc[df2['2012'] > 40, '2013')
# Give all the row values for '2012' & '2013' if the row in '2012' was greater than 40
>> display(df2.loc[df2['2012'] > 40, ['2012', '2013']])
# Give me the row values for 'Country' & '1990' for country numbers 3 & 5
>> display(df2.loc[[3,5], ['Country', '1990']])
```

Group By:
- 'groupby' method separates data into different groups based off shared characteristics
- Like generator functions & REQUIRES 'aggregations' to give an output
- DOCS: https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html
- Example:

```
>> countries.groupby('Income Group').count()
>> countries.groupby('Income Group').count().reset_index()
```

Joining Tables:

- DOCS: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html
- METHOD: dataframe.merge(right, how = 'inner', on = None, left_on = None, right_on = None, left_index = False, right_index = False, suffixes = ('_x', '_y'), copy = True, indicator = False, validate = None)
- PARAMETERS:
  - 'right' is the right table/DataFrame/Series to merge with
  - 'how' (default inner) refers the type of join: {'left', 'right', 'outer', 'inner','cross'}
  - 'on' refers to the columns/index levels to join on (must be found on BOTH DataFrames), if 'None' & not index merging default on column intersections
  - 'left_on' uses the column or index names to join on LEFT table
  - 'right_on' uses to the column or index names to join on RIGHT table
  - 'left_index' uses the index from the LEFT DataFrame as the join keys
  - 'right_index' uses the index from the RIGHT DataFrame as the join keys
  - 'sort' (default False) sorts the join keys in lexicographically (alphabetical order)
  - 'suffixes' indicates the suffix to add overlapping column names in left & right
  - 'copy' (default True) ensures that future changes to either table does NOT effect the merge results [see shallow vs copy]
  - 'indicator' (default False) adds a column with information of the source of each row (if True)
  - 'validate' (default False) checks if merge is of specified type:
    - 'one-to-one' ('1:1') checks of merge keys are unique in both LEFT & RIGHT datasets
    - 'one-to-many' ('1:m') checks if merge keys are unique in LEFT dataset
    - 'many_to_one' ('m:1') checks if merge keys are unique in RIGHT dataset



TUTORIAL 2 - VISUALISATION WITH PYTHON

- 'matplitlib' is a common 2D plotting library for data visualisation
  - DOCS: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot
    >> import matplotlib.pyplot as plt
  - Produces figures & charts on screen & in an image
  - Jupyter supports animations
- Alternatives: seaboard, pandas.plot, bokeh, folium, plotly, etc
- Complementary libraries:

```
>> import pandas as pd
>> import numpy as np
>> import IPython.display as display
```

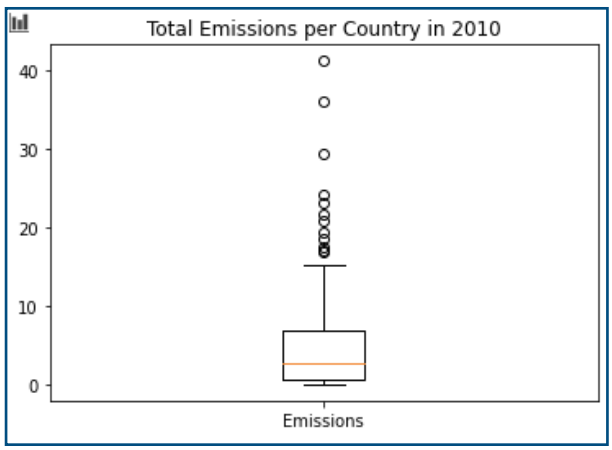
- matplotlib colours: https://matplotlib.org/stable/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py

BOX PLOTS:

- DOCS: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot
- Elements: Quartiles, IQR whiskers
- 'Box Plots' display a statistical summary of quartiles in a data set
- 'IQR' = Q3 - Q2 (Difference between quartiles at 75% & 25%)
- Whiskers are plotted at a distance 1.5 times the IQR below + above the quartiles
- Data values that fall outside the whiskers are considered outliers

Example Box Plots:

```
# Importing content from csv
>> df = pd.read_csv('data/emission.csv', encoding = 'ISO-8859-1')

# Exclude NaN values from output
>> df_2010 = df['2010'].dropna()

>> plt.boxplot(df_2010)

# Setting a title, renaming the x-axis & displaying box plot
>> plt.title("Total Emissions per Country in 2010")
>> plt.xticks([1], ["Emissions"])
>> plt.show()
```

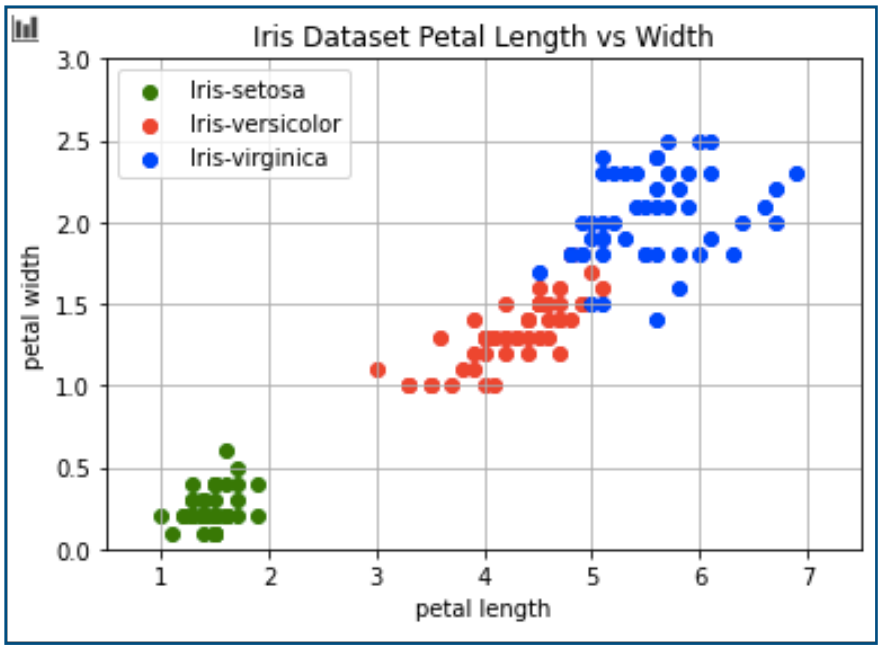



```
# Can summarise & extract statistics:
>> summary = df['2010'].dropna().describe()
>> iqr = summary.loc["75%"] - summary.loc["25%"]
>> whiskers = min(summary.loc["75%"] + 1.5*iqr, summary['max']),
max(summary.loc["25%"] - 1.5*iqr, summary['min'])
# NOTE: 2 arguments for min() & max() as these functions return the large of the 2
```

SCATTER PLOTS:
- DOCS: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html
- 'Scatter Plots' are used to display relationships between two variables (x vs y)

Example Box Plots:

```
# Extracting 5 random rows
>> df_iris = pd.read_csv('data/iris.csv',encoding = 'ISO-8859-1')
>> display(df_iris.sample(5))

# Filter/Locate the data required
>> setosa = df_iris.loc[df_iris['Name'] == 'Iris-setosa']
>> veriscolor = df_iris.loc[df_iris['Name'] == 'Iris-versicolor']
>> virginica = df_iris.loc[df_iris['Name'] == 'Iris-virginica']

# define an array of flowers
>> flowers = (setosa, veriscolor, virginica)

# define an array of different colours
>> colours = ('g', 'r', 'b')

>> for flower, c in zip(flowers, colours):
    # get the name of the flower for labels
>>     name = flower['Name'].values[0]
>>     plt.scatter(flower['PetalLength'], flower['PetalWidth'], color=c, label=name)
    # if you provide a label, then you can call plt.legend() to display the label name!

# some arguments for the overall plot
>> plt.ylabel("petal width")
>> plt.xlabel("petal length")
>> plt.title("Iris Dataset Petal Length vs Width")
>> plt.xlim(0.5,7.5)
>> plt.ylim(0,3)
>> plt.grid(True)
>> plt.legend()
>> plt.show()
```



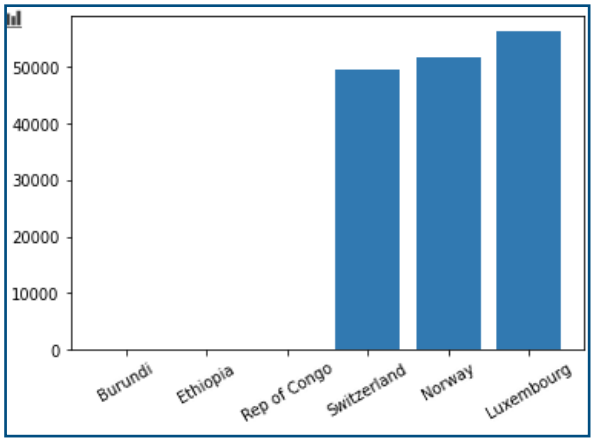
BAR CHARTS:
- DOCS: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.bar.html
- 'Bar Charts' display data as bars that are NOT touching or adjacent
- Suitable to display categorical data for comparison
- Some datasets may be better off being 'arranged' before being displayed
  - Numpy: np.arrange() is equivalent to range()
  - DOCS: https://numpy.org/doc/stable/reference/generated/numpy.arange.html
  - For large arrays use np.arrange()

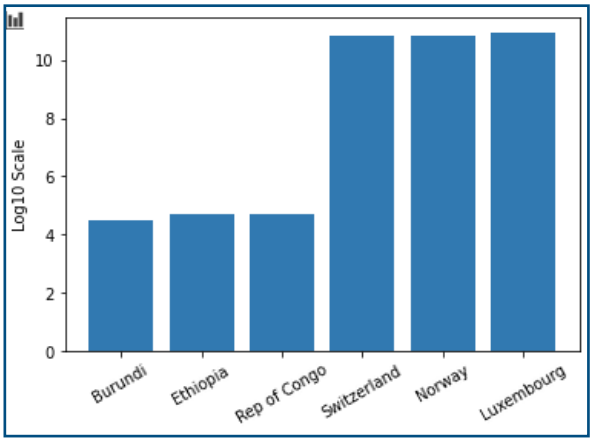
Bar Charts Example 1 - Standard

```
# Generating Data
>> countries = ['Burundi','Ethiopia','Rep of Congo','Switzerland','Norway','Luxembourg']
>> gnp = [90, 110, 110, 49600, 51810, 56380] # GNP per capita data (2004)

# Setting up the y-axis
>> plt.bar(np.arange(len(gnp)), gnp)

# Set the rotation of the x-values to 30 degrees
# For each x-val between 0 and 5, display the country name instead
>> plt.xticks(np.arange(len(countries)), countries, rotation=30)
>> plt.show()
```

Bar Charts Example 1 Remastered - Logarithmic Scale
- Large disparity between data values may result in values NOT being displayed
- 'Logarithmic Plot' can show solutions relative to each other using log
  - To get original values take the exp() (exponential) of the log scale
  - Example: log (56380) = 10.9398… & original value is exp(10.9398) = 56380
- Example Solution:

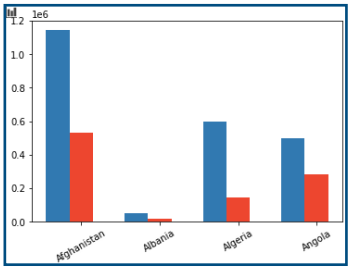
```
# Applying logarithmic operation
>> log_gnp = [np.log(i) for i in gnp]
>> plt.bar(np.arange(len(gnp)), log_gnp)

# Chanding axis labels
>> plt.xticks(np.arange(len(countries)), countries, rotation=30)
>> plt.ylabel("Log10 Scale")
>> plt.show()
```





Bar Charts Example 2 - Clustered

- 'Clustered Bar Charts' involves the inclusion of several measurements for each entity
- NOTE: matplotlib allows bars to be recoloured, & moved by certain units
- Example:



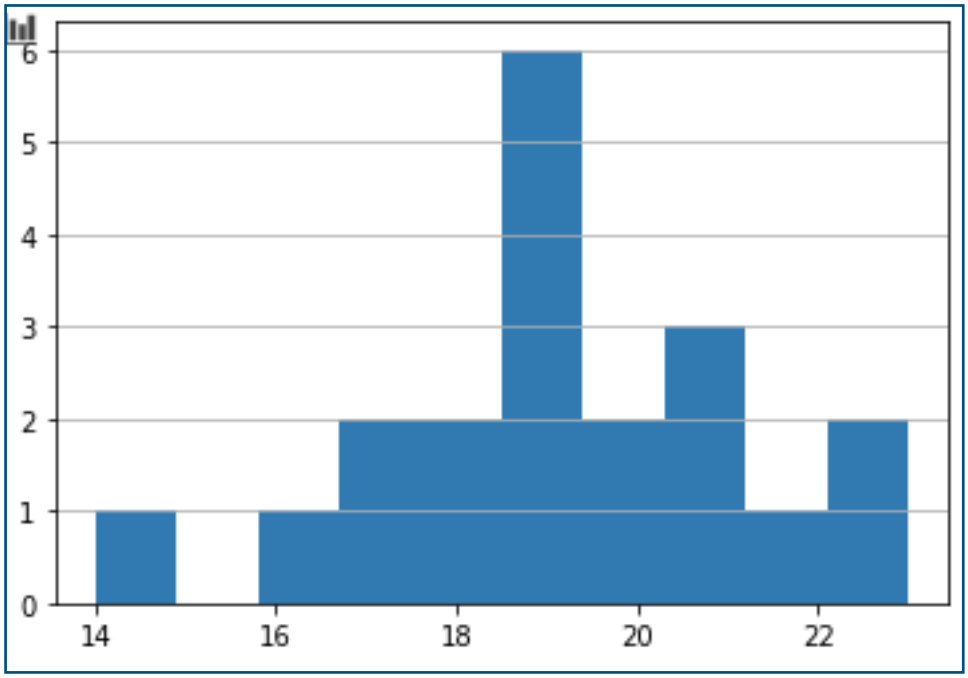

```
# Generating Data
countries = ['Afghanistan', 'Albania', 'Algeria', 'Angola']
births = [1143717, 53367, 598519, 498887]
deaths = [529623, 16474, 144694, 285380]

# -0.3 to move the 'births' bar to the left
plt.bar(np.arange(len(births))-0.3, births, width=0.3)
plt.bar(np.arange(len(deaths)), deaths, width=0.3,color='r')

# Labelling axis & displaying
plt.xticks(np.arange(len(countries)),countries, rotation=30)
plt.show()
```

HISTOGRAMS:

- DOCS: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html
- 'Histograms' are similar to bar charts except bars are touching & organised in fixed intervals
- Suitable for display non-categorical data on the x-axis

Histogram Example 1:

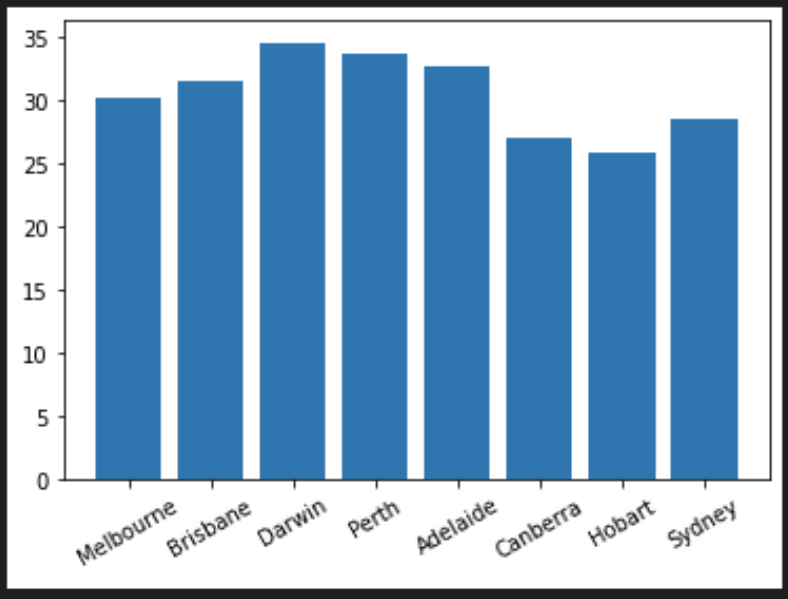
```
# Generating data & histogram
from random import choice
ages = [17,18,18,19,21,19,19,21,20,23,19,22,20,21,19,19,14,23,16,17]
plt.hist(ages, bins=10)
plt.grid(which='major', axis='y')
plt.show()
```



Histogram Example 2:

```
# Accessing csv data & extracting data
df_data = pd.read_csv("data/max_temp.csv")
display(df_data)
major_cities = [city for city in df_data["city/month"]]

# Two methods to obtain average temperature
average_temp = [df_data.loc[i].values[1:].mean() for i in range(len(df_data))]
df_data["average"] = df_data.mean(axis = 1)

# Setting axis titles & displaying graph
plt.bar(np.arange(len(major_cities)), df_data["average"])
plt.xticks(np.arange(len(major_cities)), major_cities, rotation=30)
plt.show()
```

	city/month	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	Melbourne	41.2	35.5	37.4	29.3	23.9	16.8	18.2	25.7	22.3	33.5	36.9	41.1
1	Brisbane	31.3	40.2	37.9	29.0	30.0	26.7	26.7	28.8	31.2	34.1	31.1	31.2
2	Darwin	34.0	34.0	33.2	34.5	34.8	33.9	32.0	34.3	36.1	35.4	37.0	35.5
3	Perth	41.9	41.5	42.4	36.0	26.9	24.5	23.8	24.3	27.6	30.7	39.8	44.2
4	Adelaide	42.1	38.1	39.7	33.5	26.3	16.5	21.4	30.4	30.2	34.9	37.1	42.2
5	Canberra	35.8	29.6	35.1	26.5	22.4	15.3	15.7	21.9	22.1	30.8	33.4	35.0
6	Hobart	35.5	34.1	30.7	26.0	20.9	15.1	17.5	21.7	20.9	24.2	30.1	33.4
7	Sydney	30.6	29.0	35.1	27.1	28.6	20.7	23.4	27.7	28.6	34.8	26.4	30.2



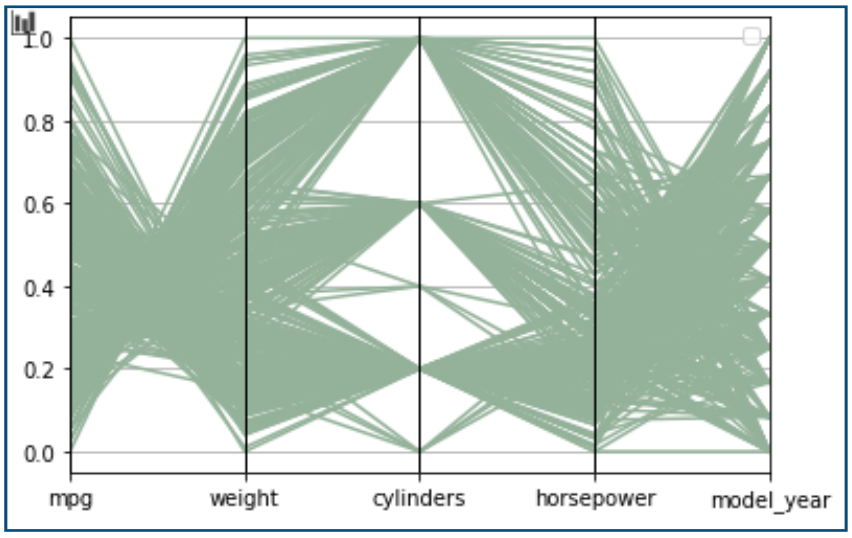
PARALLEL COORDINATES:

- DOCS: https://pandas.pydata.org/docs/reference/api/pandas.plotting.parallel_coordinates.html
- 'Parallel Coordinates' shows each row (instance) as a line & each column (feature)
- Columns are separated by fixed intervals & may need to be reordered accordingly to show a clean visualisation
- Correlations between adjacent features to be identified
- IMPORT:

```
>> from pandas.plotting import parallel_coordinates as PC
```

Parallel Coordinates Example 1:

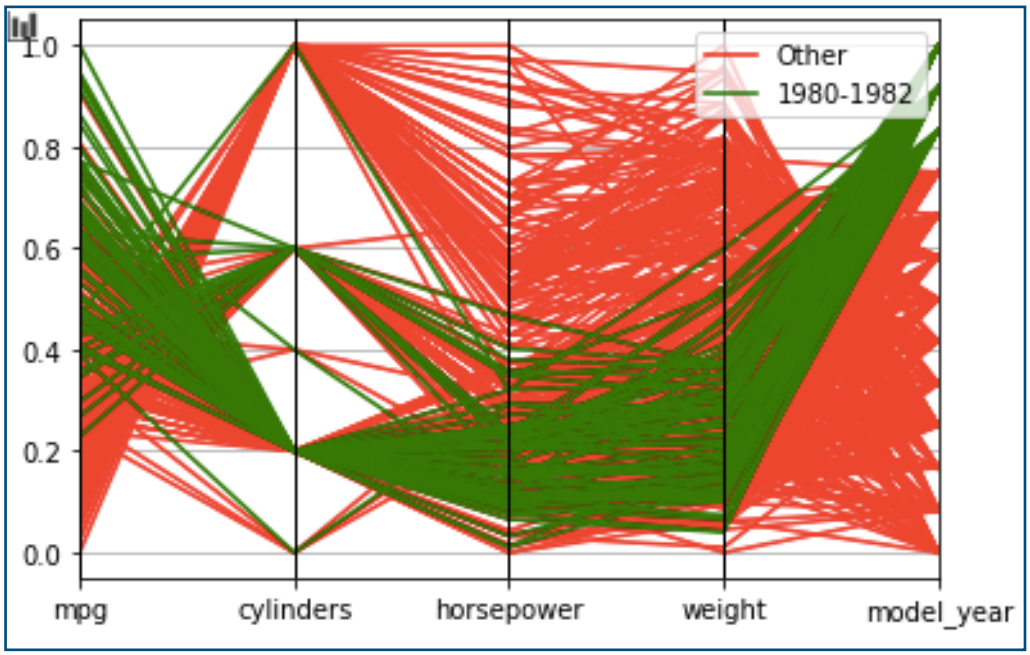
```
# Reading csv data
>> df_pc = pd.read_csv('data/mpg.csv', encoding = 'ISO-8859-1')
>> df_pc.tail()

# csv data contains a column that stores the 'name' of cars as strings
# Can reassign all rows under names as blank to remove colour on graph
>> df_pc['name'] = ''

# Normalise the data
>> cols = ('mpg', 'weight', 'cylinders', 'horsepower', 'model_year')
>> for col in cols:
        df_pc[col] = normalise(df_pc[col])
# NOTE: normalise is NOT a built in function; need to define it in program

# Selecting specific columns to display & showing graph
>> PC(df_pc[['mpg', 'weight', 'cylinders', 'horsepower', 'model_year', 'name']], 'name')
>> plt.show()
```




```
# NOTE: can utilise .sample(n) function to extract 'n' random data points
>> PC(df_pc[['mpg', 'weight', 'cylinders', 'horsepower', 'model_year', 'name']].sample(150), 'name')
```

Parallel Coordinates Example 2:

```
# Accessing csv data
df_pc = pd.read_csv('data/mpg.csv',encoding = 'ISO-8859-1')

# Make another column denoting year of model
df_pc['filter'] = df_pc['model_year'].apply(lambda x: '1980-1982' if 80 <= x <= 82 el
se 'Other')
display(df_pc)

# normalise the data again
cols = ('mpg', 'weight', 'cylinders', 'horsepower', 'model_year')
for col in cols:
    df_pc[col] = normalise(df_pc[col])

# Can show graph
PC(df_pc[['mpg','cylinders','horsepower','weight','model_year','filter']], 'filter', colo
r=["r","g"])
plt.show()
```

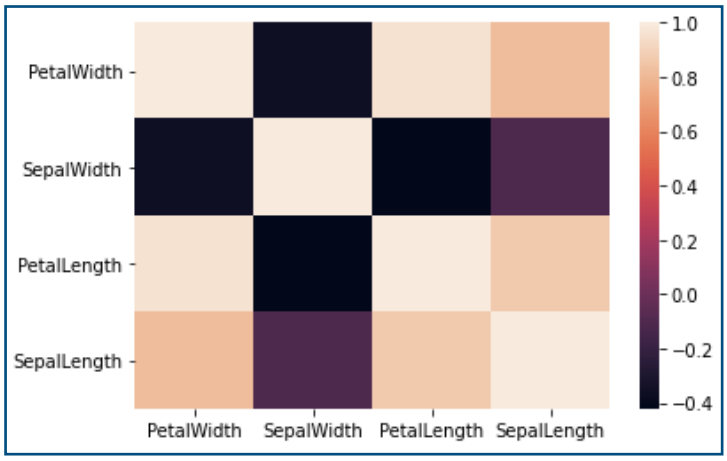


HEATMAPS:
- DOCS: https://seaborn.pydata.org/generated/seaborn.heatmap.html
  - 'Seaborn' library can be used for heatmaps
- 'Heatmaps' display aggregated information (e.g: correlation, count, etc…)
- Used to view correlation
- IMPORT:

```
>> import seaborn as sns
```

Heatmaps Example:

```
>> df_iris = pd.read_csv('data/iris.csv',encoding = 'ISO-8859-1')
>> plot_cols = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']
# cmap = colour map (colour scheme)
>> sns.heatmap(df_irsis[plotcols], cmap = 'viridis', xticklabels = True, yticklabels
= False)
>> df_iris.corr()
>> sns.heatmap(df_iris.corr())
```

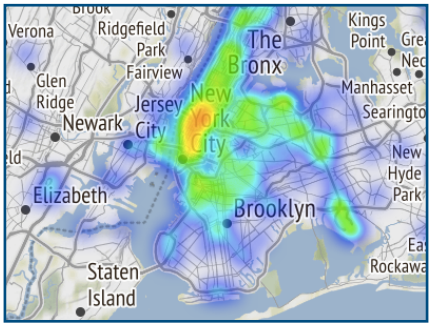


GEOSPATIAL HEATMAPS:
- DOCS: https://python-visualization.github.io/folium/plugins.html#folium-plugins
  - 'Folium' library can be used to display geospatial heatmaps
- 'Geospatial Heatmaps' shows different heated areas on a map
- IMPORT:

```
>> import folium
>> from folium.plugins import HeatMap
```

Geospatial Heatmaps Example:

```
>> COORDS = ['pickup_latitude', 'pickup_longitude']
>> nyc = folium.Map(location=[40.66, -73.94], tiles="Stamen Terrain", zoom_start=10)
>> nyc.add_child(HeatMap(df_taxi[COORDS].values, radius=10))
>> display(nyc)
```



NORMALISATION:

- 'Normalisation' is the process of scaling data to the normal distribution (or between 0 & 1)
- Allows different data to be compared through a similar distribution
- FORMULA: $x_{normalised}$ = [x - minumum(x)]/[maximum(x) - minimum(x)]
- CODE:

```
>> def normalise(data):
        """
        Function to normalise an array or series of data.
        """
        _min, _max = data.min(), data.max()
        return (data - _min) / (_max - _min)
```