INTRODUCTION TO PYTHON

VERSION 0.4

(Last edited 4/9/19)

VERSION(S) USED

Thonny
A lightweight Python viewer for beginners;
\\SYDACV01\source\Apps & Drivers\CAD\Python\Thonny

Data Camp
Online training (subscription based);
https://www.datacamp.com/

Python 3.7.3 shell
Standard deployment tool;
\\SYDACV01\source\Apps & Drivers\CAD\Python\Python Builds

How to install packages
Quick guide, use CMD prompt;
\\SYDACV01\source\Apps & Drivers\CAD\Python\Packages
Use Shift + F10, W to access folders in CMD prompt

Anaconda
\\SYDACV01\source\Apps & Drivers\CAD\Python\Anaconda

CONTENTS

GETTING STARTED

Data in python is called objects!

Immutable objects cannot be changed.

GETTING STARTED: PRINT

Print is the output operation, displays a result in the Shell results

SYNTAX

print(what to print)

SYNTAX (MULTIPLE OBJECTS)

print(a, b, c)

ARITHMETIC FUNCTIONS

Usually spaces are required to 'break' functions into pieces
e.g. (2 + 4) not (2+4)

OPERATORS - CORE

+ addition
- subtraction
* multiplication
/ division

+	addition
-	subtraction
*	multiplication
/	division

OPERATORS - OTHER

**	exponential
%	remainder

UPDATE VARIABLES VIA OPERATORS (+= ETC)

variable+=number   applies the operator to variable, then updates variable to that result

E.g.
i=1
i+=1
print(i)
i*=5
print(i)

```
>>> %Run test.py

  2
  10
```

FUNCTION: SUM

sum(x)

Note: define x as a variable prior, it cannot sum a list written in the brackets!
eg. List = (1,2,3) *then* sum(list)

LOGIC FUNCTIONS

Set Booleans using True or False (capitalized)

BOOLEANS AS INTEGERS

As an integer or sum, true = 1, false = 0

SYNTAX – SET BOOLEANS IMPLICITLY

variable = True
or
variable = False

SYNTAX – LOGIC CHECKS

x == y	x equal to y
x!=y	x not equal to y
x < y	x less than y
x > y	x greater than y
x>=y	x greater than or equal to y
x<=y	x less than or equal to y

x and y typically function best as numbers – Python can handle floats and integers as well.

Booleans can be compared to their integer representations (1 for True and 0 for False).

Surprisingly, Python can also compare strings for greater/less than checks.
It will deduce the result based on alphabetical order (earlier = smaller).

E.g.
```
print("z" > "a")
print("alicia" > "gavin")
```

```
>>> %Run test.py

  True
  False
```

SYNTAX – LOGIC CHECKING ARRAYS

We can run logic statements of Numpy arrays as well. It will return an array with all of the outcomes.

E.g.
```
import numpy as np
arr_1 = np.array([1,2,3,4,5])
print(arr_1>2)
```

```
>>> %Run test.py
  [False False  True  True  True]
```

Arrays can be also be compared to one another when of the same shape.

E.g.
```
import numpy as np
arr_1 = np.array([1,2,3,4,5])
arr_2 = np.array([5,4,3,2,1])
print(arr_1>arr_2)
```

```
>>> %Run test.py
  [False False False  True  True]
```

SYNTAX – LOGIC OPERATORS

Logic can be checked using typical operators like most programs.

Bool and Bool	Both are true
Bool or Bool	At least one is true
not(Bool)	The opposite result

SYNTAX – ANY/ALL (ARRAY)

We can run and/or functions across all values in an array like a big collection of logic statements.

np.any(condition)	At least one value in array is true
np.all(condition)	All values in array are true

E.g.
```
import numpy as np
arr_1 = np.array([1,2,3,4,5])
arr_2 = np.array([5,4,3,2,1])
print(np.any(arr_1>3))
print(np.all(arr_1>3))
```

```
>>> %Run test.py
  True
  False
```

SYNTAX – LOGIC OPERATORS (ARRAY)

Logic checking an array does not technically make sense, as each element must be checked against another array. We can check both single arrays for conditions, or multiple arrays for conditions in one statement.

np.logical_and(condition, condition)
np_logical_or(condition, condition)
np.logical_not(condition)

E.g.
```
import numpy as np
arr_1 = np.array([1,2,3,4,5])
arr_2 = np.array([5,4,3,2,1])
print(np.logical_and(arr_1>2,arr_2>1))
```

```
>>> %Run test.py
  [False False  True  True False]
```

SYNTAX – FILTER BY LOGIC (ARRAY)

We can mask conditions across arrays like we can with lists using square brackets.

Variable[condition]

E.g.
```
import numpy as np
arr_1 = np.array([1,2,3,4,5])
arr_2 = np.array([5,4,3,2,1])
print(arr_1[arr_1>3])
print()
print(arr_2[arr_2>3])
```

```
>>> %Run test.py
  [4 5]

  [5 4]
```

SYNTAX – FILTER BY LOGIC (PANDAS)

We can run logic statements on Pandas dataframes also. Typically, we isolate the column first.

Dataframe["col name"] condition

We can apply this Boolean to the dataframe in square brackets so sub-set the data.

Dataframe[Boolean series]

E.g.

```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)
print()

is_huge = data_1["area"]>8
print(is_huge)
print()

print(data_1[is_huge])
```

```
>>> %Run CSV.py

         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98

BR     True
RU     True
IN    False
CH     True
SA    False
Name: area, dtype: bool

   country   capital    area  population
BR  Brazil  Brasilia   8.516       200.4
RU  Russia    Moscow  17.100       143.5
CH   China   Beijing   9.597      1357.0
```

SYNTAX – FILTER BY AND/OR (PANDAS)

We can use the same function as numpy array logic checking on dataframes.

np.logical_and(condition, condition)
np.logical_or(condition, condition)

We can then filter by the result as per before.

E.g.

```
import pandas as pd
import numpy as np
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)
print()

is_huge = data_1["area"]>8
isnt_toobig = data_1["area"]<10

print(data_1[np.logical_and(is_huge, isnt_toobig)])
```

```
>>> %Run CSV.py

          country    capital    area  population
BR         Brazil   Brasilia   8.516      200.40
RU         Russia     Moscow  17.100      143.50
IN          India  New Delhi   3.286     1252.00
CH          China    Beijing   9.597     1357.00
SA   South Africa   Pretoria   1.221       52.98

    country   capital   area  population
BR   Brazil  Brasilia  8.516       200.4
CH    China   Beijing  9.597      1357.0
```

SYNTAX – IF/THEN/ELSE

An if statement is achieved similarly to most programs.

Note the indentation on the formula for verification, caused by the semicolon.

```
if condition:
        then result
else:
        else result
```

E.g.

```
z = 5
if z % 2 == 0:
  print("even")
else:
  print("odd")
```

```
>>> %Run test.py
  odd
```

SYNTAX – ELIF

You can add additional else/if statements using the 'elif' statement.
The final statement should still be an else statement to finish the conditions sequence.

```
if condition:
        then result
elif condition:
        then result
else:
        else result
```

E.g.

```
z = 5
if z%2 == 0:
  print("divisible by 2")
elif z%3 == 0:
  print("divisible by 3")
else:
  print("not divisible by 2 or 3")
```

```
>>> %Run test.py
  not divisible by 2 or 3
```

SYNTAX – TRY/EXCEPT

We can use this syntax to catch exceptions to an outcome, and override the result.
Often, we use this to catch errors.

ValueError
KeyError
TypeError

Try:
    function
except ValueError:
    function

E.g.

```
str_1 = 'where is wally'
str_2 = 'wenda'

try:
   result_1 = str_1.index(str_2)
except ValueError:
   result_1 = "Not Found"

print(result_1)
```

```
>>> %Run 'String working.py'
  Not Found
```

ITERATING

ITERATING/ITERABLES

Iterating is the process of repeating a process over an object in Python which is able to be iterated (iterable).
Iterables include lists, strings, sequences and more.

Under the hood, iteration is using two functions typically:

Iter() and next()

E.g. (Above the hood)
```
for char in 'word':
    print(char)
```

```
>>> %Run iter.py
  w
  o
  r
  d
```

E.g. (Below the hood)
```
word = 'word'
it = iter(word)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
>>> %Run iter.py
  w
  o
  r
  d
  Traceback (most recent call last):
    File "D:\02 Work\02.0D Crone\Python\Strings\iter.py", line 7, in <module>
      print(next(it))
  StopIteration
```

You can unpack an iterator using the star (slang term is 'splat'):

E.g.
```
word = 'word'
it = iter(word)
print(*it)
```

```
>>> %Run iter.py
  w o r d
```

SYNTAX: 'WHILE' LOOPS

A 'while' loop is essentially a repeated if statement. It is not that common, but can be very useful.
Be careful that your loop is never able to infinitely loop.

while condition:
    expression

E.g.

```
error = 50

while error > 1 :
    error /= 4
    print(error)
```

```
>>> %Run 'while loop.py'

  12.5
  3.125
  0.78125
```

We can also nest if statements within a while loop to catch conditional checks.

E.g.

```
error = 50

while error > 20 :
    if error > 30 :
        error -= 5
    else:
        error -= 3
    print(error)
```

```
>>> %Run 'while loop.py'

  45
  40
  35
  30
  27
  24
  21
  18
```

SYNTAX: 'FOR' LOOPS

To act upon all items in a list or sequence, we use a 'for' loop. Note that this is not appending onto a new list by default.

for variable **in** sequence:
  expression

E.g.

```
list_1 = ['my name is gavin', 'my name is alicia', 'my name is peter']

for sentence in list_1:
    sentence_split = sentence.split(" ")
    print(sentence_split)

print(len(sentence_split))
print(sentence_split)
```

```
>>> %Run 'String working.py'

  ['my', 'name', 'is', 'gavin']
  ['my', 'name', 'is', 'alicia']
  ['my', 'name', 'is', 'peter']
  4
  ['my', 'name', 'is', 'peter']
```

SYNTAX: 'FOR' LOOPS OVER STRING

We can use for loops to deal with a string character by character as well:

for variable **in** "string":
  expression

E.g.

```
for c in "family":
    print(c.capitalize())
```

```
>>> %Run 'while loop.py'

  F
  A
  M
  I
  L
  Y
```

SYNTAX: ENUMERATE

We can retain the index as a variable for each iteration using the following syntax:

for index, variable **in** enumerate(sequence):
    expression

E.g.
fam = [1.73, 1.68, 1.71, 1.89]

for index, height in enumerate(fam) :
    print(index, height)

```
>>> %Run 'while loop.py'

  0 1.73
  1 1.68
  2 1.71
  3 1.89
```

METHOD: 'FOR' LOOPS OVER DICTIONARY (.ITEMS)

We can use for loops to iterate over dictionary values also, but need additional syntax:

for key **in**, value in dictionary.items():
    expression

Note that dictionaries are inherently unordered, so do not expect them to follow their order.

E.g.
world = { "australia":30.55,
        "britain": 2.77,
        "USA": 38.21 }

for key, value in world.items():
    print(key+"--"+str(value))

```
>>> %Run 'while loop.py'

  australia--30.55
  britain--2.77
  USA--38.21
```

SYNTAX: 'FOR' LOOPS OVER NUMPY ARRAY (1D)

We can use for loops to iterate over 1D numpy arrays (single list) the same way we would a list:

for sub-var **in** array:
    expression

E.g.

```
import numpy as np
np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
bmi = np_weight / (np_height)**2

for val in bmi:
   print(val)
```

```
>>> %Run 'while loop.py'

  21.85171572722109
  20.97505668934241
  21.750282138093777
  24.74734749867025
  21.44127836209856
```

FUNCTION: 'FOR' LOOPS OVER NUMPY ARRAY (2D - NP.NDITER)

For multi-dimensional arrays, we need to use a function to generate a for loop:

for sub-var **in** np.nditer(array):
    expression

E.g.

```
import numpy as np
np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
meas = np.array([np_weight,np_height])

for val in np.nditer(meas):
   print(val)
```

```
>>> %Run 'while loop.py'

  65.4
  59.2
  63.6
  88.4
  68.7
  1.73
  1.68
  1.71
  1.89
  1.79
```

METHODF: 'FOR' LOOPS OVER PANDAS DATAFRAME (.ITERROWS)

If we iterate the basic way over a data frame, we only receive the header row items.

To iterate by values, we can use the syntax:

for label, row **in** dataframe.iterrows():
    expression

Note that the label is a string, the row is a pandas series.

E.g.

```
import pandas as pd
import numpy as np
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)

for label, row in data_1.iterrows():
   print(label)
   print(row)
```

```
>>> %Run CSV.py

  BR
  country          Brazil
  capital        Brasilia
  area              8.516
  population        200.4
  Name: BR, dtype: object
  RU
  country         Russia
  capital         Moscow
  area              17.1
  population       143.5
  Name: RU, dtype: object
```

We can use sub-setting functions to isolate specific results similarly:

E.g.

```
import pandas as pd
import numpy as np
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)

for label, row in data_1.iterrows():
   print(label + ": " + row["capital"])
```

```
>>> %Run CSV.py

  BR: Brasilia
  RU: Moscow
  IN: New Delhi
  CH: Beijing
  SA: Pretoria
```

SYNTAX: 'FOR' LOOPS TO NEW DATAFRAME COLUMN (+ .APPLY)

We can specify new values in a dataframe by nesting a 'loc' function within a loop.

for label, row **in** dataframe.iterrows():
    dataframe.loc[label, "column name"] = expression

Note that we are generating a new dataframe for each iteration – highly inefficient.

```
E.g.
import pandas as pd
import numpy as np
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)

for label, row in data_1.iterrows():
   data_1.loc[label, "name_length"] = len(row["country"])

print(data_1)
```

A much better workflow is to use the 'apply' function, without using a 'for' loop.

dataframe["new column"] = dataframe["sourced column"].apply(expression)

Note that if we are applying a method, we will need to also call out the class the method belongs to.

```
E.g.
import pandas as pd
import numpy as np
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)

data_1["name_length"] = data_1["country"].apply(len)
data_1["name_upper"] = data_1["country"].apply(str.upper)

print(data_1)
```

```
>>> %Run CSV.py
          country    capital    area  population  name_length    name_upper
  BR       Brazil   Brasilia   8.516      200.40            6        BRAZIL
  RU       Russia     Moscow  17.100      143.50            6        RUSSIA
  IN        India  New Delhi   3.286     1252.00            5         INDIA
  CH        China    Beijing   9.597     1357.00            5         CHINA
  SA  South Africa  Pretoria   1.221       52.98           12  SOUTH AFRICA
```

METHOD: APPEND (USING A FOR LOOP)

We can build a new list based on iterative functions by defining a new list (typically empty), then using the append method to add the resultant elements onto that list.

This method is highly useful for continual data processing (walking). Without this method, the result of a loop will typically represent the end of a function as opposed to a processing function.

List = []
for sub-variable **in** variable:
    (Apply function to sub-variable)
    List.append(sub-variable)
Print(List)

E.g.

```
import numpy as np
np.random.seed(123)
outcomes = []
tosses = range(10)
for toss in tosses:
   coin = np.random.randint(0,2)
   if coin == 0:
      outcomes.append("heads")
   else:
      outcomes.append("tails")
print(outcomes)
```

```
>>> %Run rand.py
  ['heads', 'tails', 'heads', 'heads', 'heads', 'heads', 'heads', 'tails', 'tails', 'heads']
```

DEFINING FUNCTIONS

We can define our own special functions to run over variables. In order to do this, we begin a function using 'def', then name the function and provide the positioned variables within the function in brackets. These functions can be called on later in a code so are very useful.

```
def function_name(variable, variable etc.):
        return expression
```

E.g.

```
def square(value):
   new_value = value ** 2
   print(new_value)

square(5)
```

```
>>> %Run DEF.py
  25
```

We can return the value rather than print it by using the syntax 'return' at the end of our function instead.

E.g.

```
def square(value):
   new_value = value ** 2
   return new_value

print(square(5))
```

```
>>> %Run DEF.py
  25
```

It is good practice to embed description text in triple quotations (docstrings) to describe what it is doing. These will not impact the script, but help others read and use our functions.

```
def square(value):
    """ returns the square of a value"""
    new_value = value ** 2
    return new_value

print(square(5))
```

We can return multiple values in the form of tuples.

```
E.g.
def raise_both(value1, value2):
    """ Raise value 1 to the power of value 2
    and vise versa."""

    new_value1 = value1 ** value2
    new_value2 = value2 ** value1

    new_tuple = (new_value1, new_value2)

    return new_tuple

print(raise_both(2,3))
```

```
>>> %Run DEF.py
  (8, 9)
```

Example function:

```
# Define count_entries()
def count_entries(df, col_name):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: langs_count
    langs_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over lang column in DataFrame
    for entry in col:

        # If the language is in langs_count, add 1
        if entry in langs_count.keys():
            langs_count[entry]+=1
        # Else add the language to langs_count, set the value to 1
        else:
            langs_count[entry]=1

    # Return the langs_count dictionary
    return langs_count

# Call count_entries(): result
result = count_entries(tweets_df, 'lang')

# Print the result
print(result)
```

GLOBAL/NON-LOCAL VS LOCAL SCOPE

Variables within a function are defined as local, whilst those outside are global (user defined) or built-in (python native). We can access a global variable by declaring a variable as global in our expressions.

In nested functions, we can call on upper function variables by declaring them as nonlocal. For nested functions, the lowest level of a function is searched before an outer level.

Remember it as 'LEGB':

L Local scope

E Enclosing functions

G Global

B Built-in

```
Variable2
def function_name(variable):
        global variable2
        return expression
```

E.g.

```
value2 = 10

def raise_both(value1):
    """ Raise value 1 to the power of value 2
    and vise versa."""

    global value2
    new_value1 = value1 ** value2
    new_value2 = value2 ** value1

    new_tuple = (new_value1, new_value2)

    return new_tuple

print(raise_both(2))
```

```
>>> %Run DEF.py
  (1024, 100)
```

Example of a nested function:

```
# Define three_shouts
def three_shouts(word1, word2, word3):
    """Returns a tuple of strings
    concatenated with '!!!'."""

    # Define inner
    def inner(word):
        """Returns a string concatenated with '!!!'."""
        return word + '!!!'

    # Return a tuple of strings
    return (inner(word1), inner(word2), inner(word3))

# Call three_shouts() and print
print(three_shouts('a', 'b', 'c'))
```

```
('a!!!', 'b!!!', 'c!!!')
```

```
# Define echo
def echo(n):
    """Return the inner_echo function."""

    # Define inner_echo
    def inner_echo(word1):
        """Concatenate n copies of word1."""
        echo_word = word1 * n
        return echo_word

    # Return inner_echo
    return inner_echo

# Call echo: twice
twice = echo(2)

# Call echo: thrice
thrice = echo(3)

# Call twice() and thrice() then print
print(twice('hello'), thrice('hello'))
```

```
hellohello hellohellohello
```

DEFAULT ARGUMENTS

If we want some elements to revert to a default value when not specified, we call their function variable with a formula.

E.g.

```
def power(number, pow=1):
    """Raise number to the power of pow."""
    new_value = number ** pow
    return new_value

print(power(2))
print(power(2,3))
```

```
>>> %Run DEF.py

  2
  8
```

ARGS AND KWARGS

Need to read/practice this one more – still don't get it…!

If we want to take a set of keyword/value pair arguments instead (no specific length), we can use **kwargs and a dictionary for/in key value pair loop.

E.g.

```
def printall(**kwargs):
    """Print out key-value pairs in **kwargs."""

    #print out key value pairs
    for key, value in kwargs.items():
        print(key + ": " + value)

printall(name='gavin', role='bim manager')
```

```
>>> %Run DEF.py

  name: gavin
  role: bim manager
```

LAMBDA ARGUMENTS

We can write condensed functions by specifying a function using the word lambda across one line instead

Function name = lambda variable, variable: expression.

E.g.
```
raise_to_power = lambda x,y: x**y
print(raise_to_power(2,3))
```

```
>>> %Run DEF.py
  8
```

FUNCTION: MAP

We can pass a function to all elements in a list using ‘map’ and an embedded lambda function.
Note that the resultant object will be a map, not a list. We must use the list function to convert it to a list.

Function name = map(lambda var:expression, list)
E.g.
```
nums = [1,2,3,4,5]
square_all = map(lambda num: num**2, nums)
print(square_all)
print(list(square_all))
```

```
>>> %Run DEF.py
  <map object at 0x04040790>
  [1, 4, 9, 16, 25]
```

FUNCTION: FILTER

We can pass a function to all elements in a list using ‘filter’, which will mask out all elements yielding a false statement.

Function name = filter(lambda var:expression, list)

E.g.
```
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas', 'gimli', 'gandalf']
result = filter(lambda member: len(member)>6, fellowship)
result_list = list(result)
print(result_list)
```

```
>>> %Run DEF.py
  ['samwise', 'aragorn', 'boromir', 'legolas', 'gandalf']
```

FUNCTION: REDUCE

We can pass a function to all elements in a list using 'reduce', which will roll out a function across all items in a list based on a relationship you set between two variables.

Function name = reduce(lambda var:expression, list)

E.g.

```
from functools import reduce
nums = [1,2,3,4,5,6,7,8,9,10]
result = reduce(lambda item1,item2: item1*item2, nums)
print(result)
```

```
>>> %Run DEF.py
  3628800
```

FUNCTION: HANDLING ERRORS

We can embed try/except statements in our functions to catch likely errors.

E.g.

```
def sqrt(val1):
   try:
      return val1 ** (0.5)
   except TypeError:
      return 'ERROR: Function requires a float or int input'

print(sqrt(5))
print(sqrt('text'))
```

```
>>> %Run DEF.py
  2.23606797749979
  ERROR: Function requires a float input
```

We can specify value errors with custom description also, for example square rooting negative numbers.

E.g.

```
def sqrt(x):
   if x < 0:
      raise ValueError('x must be non-negative')
   try:
      return x ** 0.5
   except TypeError:
      print('x must be an int or float')

print(sqrt('text'))
```

```
>>> %Run DEF.py
  Traceback (most recent call last):
    File "D:\02 Work\02.0D Crone\Python\Strings\DEF.py", line 9, in <module>
      print(sqrt(-5))
    File "D:\02 Work\02.0D Crone\Python\Strings\DEF.py", line 3, in sqrt
      raise ValueError('x must be non-negative')
  ValueError: x must be non-negative
```

DATA TYPES

Data types are critical to understand, can effect processing.

FUNCTION: CHECK TYPE

variable = type(other variable)
Output is in format <class 'type'>

COMMON DATA TYPES

Int	integer (aka double)
Float	number with a decimal
Bool	boolean
Str	string
List	list

FUNCTION: CONVERT DATA TYPES

str(variable)	...will convert data to strings
int(variable)	...will convert data to integers
float(variable)	...will convert data to integers
bool(variable)	...will convert data to booleans

STRINGS

'' or "" implies strings

Be careful how you encapsulate your strings when trying to use apostrophes in strings.

```
my_string = 'And this? It's the wrong string'
```

```
my_string = "And this? It's the correct string"
```

Strings can be concatenated like dynamo
Strings can also be multiplied

FUNCTION: LEN (LENGTH)

len(string)

Will return the length of a string as a float.

```
>>> %Run 'String working.py'

  3
  <class 'int'>
```

DEFINE STRINGS AS VARIABLES

str1 = 'ab'
str2 = 'cd'

FUNCTION: JOIN STRINGS

print(str1 + str2)

e.g.
str_1 = "My name is"
str_2 = "Gavin"
print(str_1 + " " +str_2)

```
>>> %Run 'String working.py'
  My name is Gavin
```

DATA TYPE: TUPLES

Tuples are sets of objects that are like lists, but are immutable (cannot be edited).
See them as value containers, which simply hold onto data in an order/packet.

We can unpack them by assigning them to variables.

E.g.
even_nums = (2,4,6)
print(type(even_nums))
a,b,c = even_nums

print(a)
print(b)
print(c)

You can also access their values like you would with lists.

E.g.
even_nums = (2,4,6)
print(type(even_nums))
a,b,c = even_nums

print(even_nums[0])
print(even_nums[1])
print(even_nums[2])

```
>>> %Run DEF.py
  <class 'tuple'>
  2
  4
  6
```

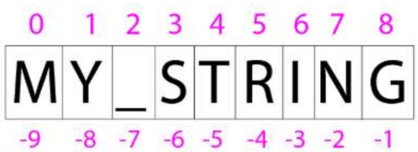
SYNTAX: INDEXING/SLICING CHARACTERS

Follow a string variable with square brackets to extract a character. This works the same way as list indexing.

E.g.
```
str_1 = "My name is Gavin"
str_2 = str_1[4]
print(str_2)
```

```
>>> %Run 'String working.py'
  My name is Gavin
>>> %Run 'String working.py'
  a
```

Work backwards from the end using negative indices, beginning at -1.



We can slice the same way as lists as well.

E.g.
```
str_1 = "My name is Gavin"
str_2 = str_1[3:7]
print(str_2)
```

```
>>> %Run 'String working.py'
  name
```

E.g.
```
str_1 = "My name is Gavin"
str_2 = str_1[:7]
print(str_2)
```

```
>>> %Run 'String working.py'
  My name
```

SYNTAX: EVERY NTH CHARACTER (STRIDE)

You can add a number after a range to introduce a stride to the returned characters.

E.g.
str_1 = "My name is Gavin"
str_2 = str_1[0:8:2]
print(str_2)

My name is Gavin

```
>>> %Run 'String working.py'
  M ae
```

SYNTAX: REVERSE STRING

There is a special syntax for reversing a string.

string[::-1]

E.g.
str_1 = "My name is Gavin"
str_2 = str_1[::-1]
print(str_2)

```
>>> %Run 'String working.py'
  nivaG si eman yM
```

VERIFYING PALINDROMES

We can identify palindromes using simple 'if' logic checks.

```
E.g.
str_1 = 'glenelg'
str_2 = str_1[::-1]

if str_1 == str_2:
   print('yes')
else:
   print('no')
```

```
>>> %Run 'String working.py'
  yes
```

METHOD: (R)SPLIT (INTO SUBSTRINGS)

There are a few ways to split strings using method chaining.

variable.split(sep=”string”)
Will split a string, defined by the separating string.

If you are splitting using spaces, you do not need to specify the separator character, just use empty brackets.

E.g.
str_1 = "My name is Gavin"
str_2 = str_1.split(sep=" ")
print(str_2)

```
>>> %Run 'String working.py'
  ['My', 'name', 'is', 'Gavin']
```

variable.split(sep=”string”, maxsplit=X)

Will split a string, but only split up to the specified number of times.
The final sub-string will be the remainder with no splits.

E.g.
str_1 = "My name is Gavin"
str_2 = str_1.split(sep=" ", maxsplit=2)
print(str_2)

```
>>> %Run 'String working.py'
  ['My', 'name', 'is Gavin']
```

variable.rsplit(sep=”string”, maxsplit=X)

Will split a string, but only split up to the specified number of times.
The final sub-string will be the remainder with no splits.

This splits from the right vs. the left, but keeps substrings in forward order.

E.g.
str_1 = "My name is Gavin"
str_2 = str_1.split(sep=" ", maxsplit=2)
print(str_2)

```
>>> %Run 'String working.py'
  ['My', 'name', 'is Gavin']
```

SYNTAX: ESCAPE SEQUENCES

Some character combinations can be inserted or managed to control line breaks and returns.

We can achieve the functions implied by these breaks using particular methods, they do not trigger by simple being a part of a string, so don't worry if you don't actually want to use them!

\n   new line
\r   carriage return

E.g.
This string will be split\nin two

Becomes

This string will be split
In two

METHOD: SPLIT LINES

This method can be used to split strings at the /n character.
Note that this will not include the /n character in the output.

E.g.
```
str_1 = "My name\nis Gavin"
str_2 = str_1.splitlines()
print(str_2)
```

```
>>> %Run 'String working.py'
  ['My name', 'is Gavin']
```

METHOD: JOIN (LIST OF STRINGS)

We can also put lists back together.

string.join(list of strings)

E.g.
```
list_1 = ["My", "name", "is", "Gavin"]
str_1 = " ".join(list_1)
print(str_1)
```

```
>>> %Run 'String working.py'
  My name is Gavin
```

METHOD: (R/L)STRIP

Strip can be used for many functions, by default it removes white space or escape sequences to left or right.

string.strip()

E.g.
str_1 = "   \rThis string will be stripped   \n"
str_2 = str_1.strip()
print(str_2)

```
>>> %Run 'String working.py'
  This string will be stripped
```

We can specify characters within the brackets to remove specific characters from the string.
However
It will still only strip explicitly from the right or left side, not within a string.

E.g.
str_1 = "This string will be stripped"
str_2 = str_1.strip('will be stripped')
print(str_2)

```
>>> %Run 'String working.py'
  This string
```

We can alternatively use the methods l/r strip to strip from one end only.

E.g.
str_1 = "This is This"
str_2 = str_1.lstrip('This')
str_3 = str_1.rstrip('This')
print(str_2)
print(str_3)

```
>>> %Run 'String working.py'
   is This
  This is
```

METHOD: FIND (SUBSTRING)

This can be used to report indices of items within a string.

string.find(substring, start, end)

Note that start and end characters are optional.
If there is no occurrence, it will return -1.

```
E.g.
str_1 = 'Where is Wally?'
there_he_is = str_1.find("Wally")
there_she_is = str_1.find("Wenda")
print(there_he_is)
print(there_she_is)
```

```
>>> %Run 'String working.py'
  9
  -1
```

METHOD: COUNT (SUBSTRING)

This method will return the number of times a substring occurs in a string.

String.count(substring, start, end)

Note that start and end characters are optional.

```
E.g.
str_1 = 'wally, wally, wally, wally'
str_2 = str_1.count('wally')
print(str_2)
```

```
>>> %Run 'String working.py'
  4
```

METHOD: REPLACE (SUBSTRING)

This method will replace occurrences of a substring.

String.replace(old, new, count)

Note the count is optional, and will specify how many times you want to replace.

E.g.
```
str_1 = 'wally, wally, wally, wally'
str_2 = str_1.replace('wally', 'wenda', 2)
print(str_2)
```

```
>>> %Run 'String working.py'
  wenda, wenda, wenda, wenda
```

SYNTAX: POSITIONAL FORMATTING

Positional formatting allows us to create text structures with 'blanks' to fill with data.
The placed text is held in the format via placeholders, implied by curly braces/brackets.

'text {} text {}'.format(value, value)

E.g.
```
x = 'which'
y = 'formatted'
format_1 = 'this is a string {} has been positionally {}'.format(x,y)
print(format_1)
```

```
>>> %Run 'string format.py'

  this is a string which has been positionally formatted
```

We can iterate lists and apply positional formatting for effective outcomes:

E.g.
```
x = ['which', 'that']
y = ['formatted', 're-formatted']
string_m1 = 'this is a string {} has been positionally {}'

for i in range(2):
   a = x[i]
   b = y[i]
   print(string_m1.format(a,b))
```

```
>>> %Run 'string format.py'

  this is a string which has been positionally formatted
  this is a string that has been positionally re-formatted
```

We can specify the indices of placement by embedding numbers in the braces:

E.g.
```
string_m1 = '{2}, {0}, buckle my {1}'
print(string_m1.format('two', 'shoe', 'one'))
```

```
>>> %Run 'string format.py'

  one, two, buckle my shoe
```

We can specify named placeholders and draw data from such elements as dictionaries also:

E.g.
combat_1 = {"spell":"fireball", "target":"kobold", "damage":9001}
damage_log = '{data[spell]} hits {data[target]} for {data[damage]} damage!'

print(damage_log.format(data=combat_1))

```
>>> %Run 'string format.py'
  fireball hits kobold for 9001 damage!
```

We can also specify the format of data to be presented using format specifiers:

E.g.
combat_1 = {"spell":"fireball", "target":"kobold", "damage":9001}
damage_log = '{data[spell]} hits {data[target]} for {data[damage]:f} damage!'

print(damage_log.format(data=combat_1))

```
>>> %Run 'string format.py'
  fireball hits kobold for 9001.000000 damage!
```

Adding a decimal with a number will imply the significant figures to retain in the output:

E.g.
combat_1 = {"spell":"fireball", "target":"kobold", "damage":9001}
damage_log = '{data[spell]} hits {data[target]} for {data[damage]:.2f} damage!'

print(damage_log.format(data=combat_1))

```
>>> %Run 'string format.py'
  fireball hits kobold for 9001.00 damage!
```

FUNCTION: DATETIME.NOW

Date times are always important to format, we can call the time now using:

datetime(now)
Note that datetimes are from the datetime package.

E.g.
from datetime import datetime
print(datetime.now())

```
>>> %Run 'string format.py'

  2019-09-07 17:08:10.818512
```

SYNTAX: FORMATTING DATETIMES

We can use formatting functions to modify datetime output:

E.g.
from datetime import datetime
get_date = datetime.now()
message = "Good morning. Today is {today:%B %d, %Y}. It's {today:%H:%M} ... time to work!"
print(message.format(today=get_date))

```
>>> %Run 1.py

  Good morning. Today is September 10, 2019. It's 00:12 ... time to work!
```

SYNTAX: DATETIME FORMAT CODES

There are a lot…. But here they are!

Directive	Meaning	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...
%w	Weekday as a decimal number.	0, 1, ..., 6
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99
%-y	Year without century as a decimal number.	0, 1, ..., 99
%Y	Year with century as a decimal number.	2013, 2019 etc.
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%p	Locale’s AM or PM.	AM, PM
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%z	UTC offset in the form +HHMM or -HHMM.	
%Z	Time zone name.	
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366

Directive	Meaning	Example
%U	Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, …, 53
%W	Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, …, 53
%c	Locale's appropriate date and time representation.	Mon Sep 30 07:06:05 2013
%x	Locale's appropriate date representation.	09/30/13
%X	Locale's appropriate time representation.	07:06:05
%%	A literal '%' character.	%

Source: https://www.programiz.com/python-programming/datetime/strftime

FORMATTED STRING LITERAL

Also known as ‘f-strings’, these are a more recent and useful syntax to format strings vs. positional formatting.

f”this is a {variable} and this is {another}”

E.g.
unit1 = 'wizard'
unit2 = 'kobold'
spell1 = 'fireball'
damage1 = 9000

print(f"{unit1} hits {unit2} with a {spell1} for {damage1:.2f} damage!")

```
>>> %Run 'string literal.py'

  wizard hits kobold with a fireball for 9000.00 damage!
```

F-strings are also helpful because you can nest statements within the placeholder brackets.
E.g.
unit1 = 'wizard'
unit2 = 'kobold'
spell1 = 'fireball'
damage1 = 9000

print(f"{unit1.title()} hits {unit2} with a {spell1.upper()} for {damage1:.2f} damage!")

```
>>> %Run 'string literal.py'

  Wizard hits kobold with a FIREBALL for 9000.00 damage!
```

F-strings also support additional type conversion options:

!s	string version
!r	printable representation (‘string’)
!a	!r but escapes ascii characters
e	scientific notation
d	integer/digit
f	float

E.g.
…
print(f"{unit1.title()} hits {unit2!r} with a {spell1.upper()!a} for {damage1:.2f} damage!")

```
>>> %Run 'string literal.py'

  Wizard hits 'kobold' with a 'FIREBALL' for 9000.00 damage!
```

Datetime is similar to before:
E.g.
from datetime import datetime
my_today = datetime.now()
print(f"Today's date is {my_today:%B %d, %Y}")

```
>>> %Run 'string literal.py'

  Today's date is September 10, 2019
```

Dictionaries are handled similarly, but we put the key in quotes unlike positional format:

f"this is how you call out a {dictionary['key']}"

E.g.
family = {"dad":"Gavin", "siblings":"Alicia"}
print(f"Is your sister called {family['siblings']}?")

```
>>> %Run 'string literal.py'

  Is your sister called Alicia?
```

FORMATTED STRINGS BY TEMPLATE

The template method is much slower, but better at handling unformatted data types coming from an external source. We must load the Template module of the string class prior to running this function.

From string import Template
Variable = Template('$placeholder has been $placeholder2')
Print(Variable.substitute(placeholder = this, placeholder2 = has been enlightening))

Note that placeholders are defined using a dollar sign, and need to be specified via a substitution method from there.

E.g.
from string import Template
job = "Data Science"
name = "sexiest job of the 21st century"
my_string = Template('$title has been called $description')
print(my_string.substitute(title=job, description=name))

```
>>> %Run 'string literal.py'

  Data Science has been called sexiest job of the 21st century
```

Enclose variables in curly braces if we want to join additional text prior to a space.

E.g.
from string import Template
my_string = Template('I find Python very ${noun}ing, but my office has no $noun!')
print(my_string.substitute(noun="interest"))

```
>>> %Run 'string literal.py'

  I find Python very interesting, but my office has no interest!
```

A dollar sign can be escaped as a prefix using two sequential dollar signs:

E.g.
from string import Template
my_string = Template('It only cost $$$price!')
print(my_string.substitute(price=21.54))

```
>>> %Run 'string literal.py'

  It only cost $21.54!
```

METHOD: SAFE SUBSTITUTE

We run the risk with dictionaries and similar data types that we run out of values t

E.g.
from string import Template
favourite = dict(flavor="chocolaye")
my_string = Template('I love $flavor $cake so much!')
print(my_string.substitute(favourite))

```
>>> %Run 'string literal.py'
  Traceback (most recent call last):
    File "D:\02 Work\02.0D Crone\Python\Strings\string literal.py", line 6, in <module>
      print(my_string.substitute(favourite))
    File "C:\Users\Gavin Crump\AppData\Local\Programs\Thonny\lib\string.py", line 132, in substitut
  e
      return self.pattern.sub(convert, self.template)
    File "C:\Users\Gavin Crump\AppData\Local\Programs\Thonny\lib\string.py", line 125, in convert
      return str(mapping[named])
  KeyError: 'cake'
```

We can use safe substitute to counteract this. Unmatched placeholders will be retained with their dollar prefix.

E.g.
from string import Template
favourite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake so much!')
print(my_string.safe_substitute(favourite))

```
>>> %Run 'string literal.py'
  I love chocolate $cake so much!
```

FUNCTION: REPEAT STRINGS X TIMES

print(str1 * 5)
Output will be ababababab

METHODS: CASE MANAGEMENT

String case management is achieved via chaining a method to a string.

Variable.upper()
Variable.lower()
Variable.capitalize()
Variabe.title()

REGULAR EXPRESSIONS (REGEX)

Regular expressions are used to identify and return text patterns, taking into account normal and special characters, as well as the position of text. It is called with the syntax r', and meta characters are typically called with back slashes (\).

r'st\d\s\w{3,10}'

Examples of what they are used for:

- Find and replace text
- Validate strings
- Password matching and rules

Regex is handled using the re module, typically imported with no alias.

METHOD: .FINDALL

We use this method to find and isolate occurrences within a string.

re.findall(r"match", "string")

E.g.

```
import re
regex_1 = re.findall(r"day", "today was a good day")
print(regex_1)
print(type(regex_1))
```

```
>>> %Run 'string literal.py'
  ['day', 'day']
  <class 'list'>
```

METHOD: .SPLIT

We use this method to return all the characters around each match.

re.split(r"match", "string")

E.g.

```
import re
regex_1 = re.split(r"day", "today was a good day")
print(regex_1)
print(type(regex_1))
```

```
>>> %Run 'string literal.py'
  ['to', ' was a good ', '']
  <class 'list'>
```

METHOD: .SUB

We use this method to replace all matches with another

re.sub(r“match”, “new”, “string”)

```
E.g.
import re
regex_1 = re.sub(r"day","morrow","today was a good day")
print(regex_1)
print(type(regex_1))
```

```
>>> %Run 'string literal.py'

  tomorrow was a good morrow
  <class 'str'>
```

METHOD: .SEARCH VS .MATCH

We use this method to determine if the regex is satisfied anywhere in the string.

re.search(r“regex”, “string”)

We use this method to determine if the regex is satisfied at the beginning of the string.

re.match(r“regex”, “string”)

```
import re
string= '123abc'

if re.match(r"[a-z]+", string):
   print('true')
else:
   print('false')

if re.search(r"[a-z]+", string):
   print('true')
else:
   print('false')
```

```
>>> %Run 'string literal.py'

  false
  true
```

INTRODUCING METACHARACTERS

Metacharacters imply a character or zone of a string that is required to match a condition vs. explicit text.

\w any character (word)
\W non-character (e.g. symbols)
\d digit
\D non-digit
\s white space
\S non-white space

E.g.

```
import re
regex_1 = re.findall(r"User\d","The winners are User9, UserN, User8")
print(regex_1)
print(type(regex_1))
```

```
>>> %Run 'string literal.py'

  ['User9', 'User8']
  <class 'list'>
```

REPEATED CHARACTERS (QUANTIFIERS)

Quantifiers specify how many times a metacharacter is required to its left. It is specified in curly braces.

E.g.

```
import re
passwords = ["password1234", "password5678", "pass1234"]

for password in passwords:
   print(re.search(r"\w{8}\d{4}", password))
```

```
>>> %Run 'string literal.py'

  <re.Match object; span=(0, 12), match='password1234'>
  <re.Match object; span=(0, 12), match='password5678'>
  None
```

REGEX METACHARACTERS

There are some special quantifiers:

+	Once or more times
?	Zero times or once (no more than)
*	Zero or more times (pseudo wildcard)
{n, m}	n times at least, m times at most
{n, }	n times at least
{ ,m}	m times at most
.	Matches any character (except newline)
^	Only if it is at the beginning of the string
$	Only if it is on the end of the string (on end of regex)

We can escape metacharacters by preceding them with a \

|	Or operator (regex will check either side of this twice)

E.g.

```
import re
string = 'elephants are Elephants'
print(re.findall(r"Elephant|elephant", string))
```

```
>>> %Run 'string literal.py'
  ['elephant', 'Elephant']
```

[a-zA-Z]	Any lower or upper-case characters

E.g.

```
import re
list = "My friend list: Mary24, ClaRY44, Fred3"
print(re.findall(r"[a-zA-Z]+\d+", list))
```

```
>>> %Run 'string literal.py'
  ['Mary24', 'ClaRY44', 'Fred3']
```

[ ]	Any characters found in the brackets

E.g.

```
import re
statement = 'My&name#is%Gavin. I@live in#Sydney'
print(re.sub(r"[&#@%]", " ", statement))
```

```
>>> %Run 'string literal.py'
  My name is Gavin. I live in Sydney
```

^ in square brackets reverses the regex expectations.

[^0-9]	Does not contain any numbers
[^a-zA-Z]	Does not contain any letters

EXAMPLE: EMAIL/PASSWORD VALIDITY

```
regex = r"[A-Za-z0-9!#%&*\$\.]+@\w+\.com"

for example in emails:
   if re.match(regex, example):
         print("The email {email_example} is a valid email".format(email_example=example))
   else:
         print("The email {email_example} is invalid".format(email_example=example))
```

```
The email n.john.smith@gmail.com is a valid email
The email 87victory@hotmail.com is a valid email
The email !#mary-=@msca.net is invalid
```

```
regex = r"[0-9a-zA-Z*#$%!&.]{8,20}"

for example in passwords:
   if re.search(regex, example):
         print("The password {pass_example} is a valid password".format(pass_example=example))
   else:
         print("The password {pass_example} is invalid".format(pass_example=example))
```

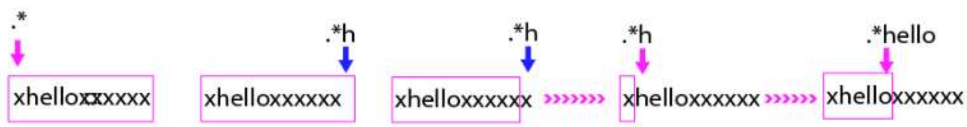
```
The password Apple34!rose is a valid password
The password My87hou#4$ is a valid password
The password abc123 is invalid
```

LAZY/NON-GREEDY MATCHING

Previous quantifiers are 'greedy', they will work towards the end of a string until they find the maximum match possible.

```
import re
re.match(r".*hello", "xhelloxxxxxx")
```

```
<_sre.SRE_Match object; span=(0, 6), match='xhello'>
```



Lazy quantifiers will return the shortest possible match. If we use a ? after our quantifier, it will instead deal with the least possible of matched characters in a + check for example.

```
import re
re.match(r".*?hello", "xhelloxxxxxx")
```

```
<_sre.SRE_Match object; span=(0, 6), match='xhello'>
```

Sometimes this is necessary so we can catch wildcard statements to the shortest possible length.

GROUPING AND CAPTURING

We can return a selection of a phrase based on matching by using parenthesis around the portion we wish to isolate. The regex will still find the overall phrase match, but only return that which we specify.

E.g.

```
import re
string_1 = 'Gavin has 4 marbles, Adam has 2 marbles and you have 1 marble.'
regex = (r'[a-zA-Z]+\s\w+\s\d+\s\w+')
regex_grp = (r'([a-zA-Z]+)\s\w+\s\d+\s\w+')
regex_grp2 = (r'[a-zA-Z]+\s\w+\s(\d+)\s\w+')

print(re.findall(regex, string_1))
print(re.findall(regex_grp, string_1))
print(re.findall(regex_grp2, string_1))
```

```
>>> %Run 'string literal.py'
 ['Gavin has 4 marbles', 'Adam has 2 marbles', 'you have 1 marble']
 ['Gavin', 'Adam', 'you']
 ['4', '2', '1']
```

We can determine multiple groups by capturing multiple portions in brackets.

E.g.

```
import re
string_1 = 'Gavin has 4 marbles, Adam has 2 marbles and you have 1 marble.'
regex = (r'[a-zA-Z]+\s\w+\s\d+\s\w+')
regex_grp = (r'([a-zA-Z]+)\s\w+\s(\d+)\s(\w+)')

print(re.findall(regex, string_1))
print(re.findall(regex_grp, string_1))
```

```
>>> %Run 'string literal.py'
 ['Gavin has 4 marbles', 'Adam has 2 marbles', 'you have 1 marble']
 [('Gavin', '4', 'marbles'), ('Adam', '2', 'marbles'), ('you', '1', 'marble')]
```

We can also subset this data in the same manner as lists:

E.g.

```
…
data = re.findall(regex_grp, string_1)
print(data[0])
print(data[0][1])
names = []
for item in data:
   names.append(item[0])
print(names)
```

```
>>> %Run 'string literal.py'
 ('Gavin', '4', 'marbles')
 4
 ['Gavin', 'Adam', 'you']
```

You can include quantifiers inside or outside the braces. If you include them within, you will get the match in its entirety (of what the quantifier detects). If it is outside, you will only receive the first occurrence the quantifier detects, even if a quantifier such as '+' is being used.

E.g.

```
import re
string_1 = 'My user name is 3e4r5fg'
regex_grp = (r'(\d[a-zA-Z]+)')
regex_grp2 = (r'(\d[a-zA-Z])+')

data = re.search(regex_grp, string_1)
data2 = re.search(regex_grp2, string_1)

print(data)
print(data2)
```

```
>>> %Run 'string literal.py'
  <re.Match object; span=(16, 18), match='3e'>
  <re.Match object; span=(16, 22), match='3e4r5f'>
```

ALTERING AND NON-CAPTURING GROUPS

Capturing 'or' statements for grouping is also possible:

E.g.

```
import re
string_1 = 'I have 1 dog, 2 cats and 3 birds'
data = re.findall(r"(\d)+\s(dog\w*|cat\w*|bird\w*)", string_1)
print(data)
```

```
>>> %Run 'string literal.py'
  [('1', 'dog'), ('2', 'cats'), ('3', 'birds')]
```

We can specify non-capturing groups using ?: at the front of the parenthesis, which is often needed for 'or':

E.g.

```
import re
string_1 = 'Today is 23rd May 2019, Tomorrow is 24th May 2019'
data = re.findall(r"(\d+)(?:th|rd)", string_1)
data1 = re.findall(r"(\d+)(th|rd)", string_1)
print(data)
print(data1)
```

```
>>> %Run 'string literal.py'
  ['23', '24']
  [('23', 'rd'), ('24', 'th')]
```

METHOD: .GROUP

We can retrieve the items of a statement using the group function.

Variable.group(Index)

If we want all items together (including what lies between), we can specify group 0.

E.g.

```
import re
string_1 = 'Python 3.0 was released on 12--03-2008'
info = re.search(r'(\d{1,2})--(\d{2})-(\d{4})', string_1)

print(info.group(1))
print(info.group(2))
print(info.group(3))
print(info.group(0))
```

```
>>> %Run 'string literal.py'

  12
  03
  2008
  12--03-2008
```

We can name and retrieve groups using the syntax ?P<name> at the front of our groups.

E.g.

```
import re
string_1 = 'Waterloo, 2017'

info = re.search(r"(?P<city>[a-zA-Z]+).*?(?P<zip>\d{4})", string_1)

print(info.group('city'))
print(info.group('zip'))
print(info.group(0))
```

```
>>> %Run 'string literal.py'

  Waterloo
  2017
  Waterloo, 2017
```

SYNTAX: BACKREFERENCE FOR REPETITION

We can backreference an element by including the syntax '\x' on the end of the regex, where x is the number of repetitions of the format we are searching for.

E.g.

```
import re
string_1 = 'It will be a great great day today.'
regex = r"(\w+)\s\1"
fixed = re.sub(regex, r"\1", string_1)
print(fixed)
```

```
>>> %Run 'string literal.py'
  It will be a great day today.
```

Back referenced elements can also be assigned to a group:

```
import re
string_1 = 'Your new code number is 23434. Please, enter 23434 to open the door.'
regex = r"(?P<code>\d{5}).*?(?P=code)"
data = re.findall(regex, string_1)
print(data)
```

```
>>> %Run 'string literal.py'
  ['23434']
```

We can then use another syntax to replace the two occurrences and replace them with a single one:

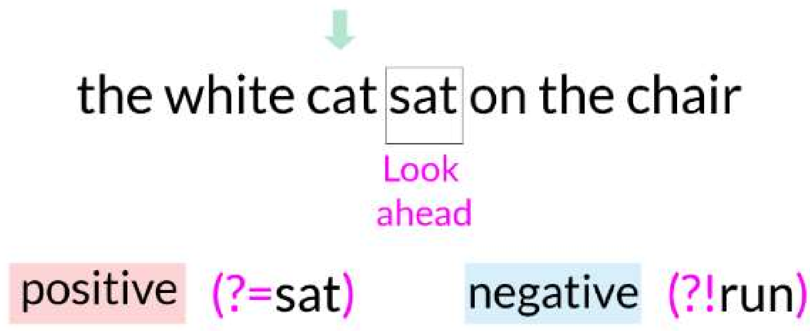
E.g.

```
import re
string_1 = 'Your new code number is 23434. Please, enter 23434 to open the door.'
regex = r"(?P<code>\d{5}).*?(?P=code)"
data = re.sub(regex, r"\g<code>", string_1)
print(data)
```

```
>>> %Run 'string literal.py'
  Your new code number is 23434 to open the door.
```

SYNTAX: LOOK AHEAD

These allow us to confirm that a sub-pattern is ahead or behind our main pattern.



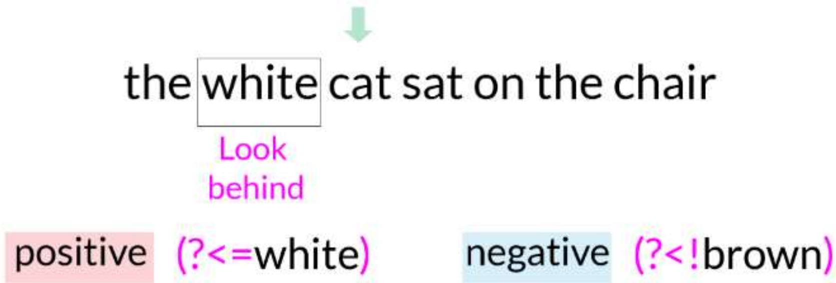
E.g.

```
import re
string_1 = 'file1.txt transferred, file2.txt transferred, file3.txt error'
regex = r"\w+\.txt(?=\stransferred)"
regex_e = r"\w+\.txt(?!\stransferred)"
data = re.findall(regex, string_1)
data2 = re.findall(regex_e, string_1)
print(data)
print(data2)
```

```
>>> %Run 'string literal.py'
  ['file1.txt', 'file2.txt']
  ['file3.txt']
```

SYNTAX: LOOK BEHIND

These allow us to confirm that a sub-pattern is ahead or behind our main pattern.



E.g.

```
import re
string_1 = 'transferred file1.txt, transferred file2.txt, error file3.txt'
regex = r"(?<=transferred\s)file\w+\.txt"
regex_e = r"(?<!transferred\s)file\w+\.txt"
data = re.findall(regex, string_1)
data2 = re.findall(regex_e, string_1)
print(data)
print(data2)
```

```
>>> %Run 'string literal.py'
  ['file1.txt', 'file2.txt']
  ['file3.txt']
```

LISTS

Implied by the use of square brackets, e.g. [a, b, c, etc.]

FUNCTION: ITEM AT INDEX

variable[X]
where X is the list index (0 through to n)

You can index from the back of a list using negative indexes
e.g. index -1 is the last item in a list

FUNCTION: LIST SLICE (BETWEEN)

Ranges can be taken using syntaxt of List[a:c], which takes a through to b
The slice will not be inclusive of b, but a is included

FUNCTION: LIST SLICE (BEFORE/AFTER)

If you leave out either side of the range you only take before or after;
List[b:] takes all from back
List[:c] takes all from front, ends 1 before c

EXAMPLE: BUILDING LISTS

```
fam1 = [["Gavin", 1.9],
        ["Alicia", 1.8],
        ["Julie", 1.7],
        ["Peter", 1.5]]

heights = [1.9, 1.8, 1.5, 1.7]
names = ["Gavin", "Alicia", "Peter", "Julie"]

fam2 = [heights, names]

print(fam1)
print(fam2)
```

FUNCTION: ITEM IN SUBLIST, AT INDEX

list[X][Y] (index Y in sub list at index X)

FUNCTION: REPLACE ITEM AT INDEX

List[X] = Y (replace item at index X with Y)

FUNCTION: REPLACE ITEMS BETWEEN INDICES

List[X:Y] = [A, B] (replace items between indices with A,B)

FUNCTION: REMOVE ITEM AT INDEX

del(list[X]) removes item at index X in list

EXAMPLE – RECURSIVE VARIABLES AND LISTS

If you point a variable to a list, it is still the same list, both references would be the same essentially. For example;

x = [1,2,3]
y = x
dely(y[0])
print(x)
would show
[2,3]

FUNCTIONS

Storing variables using functions, syntax varies, but is commonly;
functionname(input)

They are reusable code packages, like nodes/blocks
E.g. 'type' is a function

FUNCTION: MIN/MAX ITEM

max(list)
min(list)

Can be set to a variable as well
e.g. Tallest = max(height)

FUNCTION: ROUND (0DP/PRECISION)

round(x, y) *or* round(x)
if no precision, it rounds to integer
x is the list/data
y is precision of decimal

FUNCTION: HELP

help(x) provides help on the function called 'x'

FUNCTION: LENGTH (AKA COUNT)

len(x) provides the length/count of a list or string

FUNCTION: IMAGINARY NUMBERS

complex(x, y) provides a real/imaginary hybrid where x is real, y is imaginary (Xj)
e.g. (2,0) = 2
(2,2) = 2 + 2j
(0,2) = 2j

FUNCTION: SORT

sorted(x, y, reverse = bool)
x is list, y is key, reverse is true or false

EXAMPLE – SORTING A LIST

```
# Create lists first and second
first = [11.25, 18.0, 20.0]
second = [10.75, 9.50]

# Paste together first and second: full
full = first+second
print(full)

# Sort full in descending order: full_sorted
full_sorted = sorted(full, reverse = True)

# Print out full_sorted
print(full_sorted)
```

METHODS

Like a sub-class of functions, make functions look more like a 'package' of tools to expand functions.

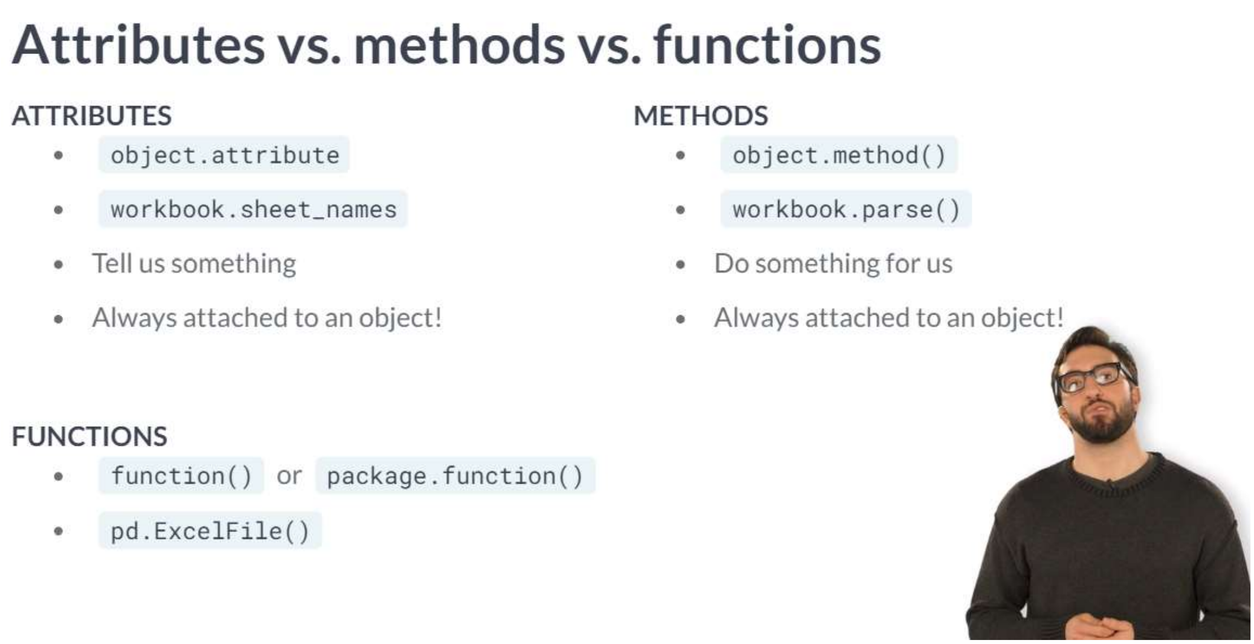
If you combine them with their data type they act as additional functions.
Some methods are used for different data types, but behave differently.

Be careful, some methods change the object they're called on and can change data types.
Use list(x) to apply functions when in doubt

Added to the list or variable using a full stop (.)

ATTRIBUTES

Methods are a sub-class of functions; attributes are a sub-class of objects. They are accessed via the same type of dot syntax. Attributes do not require parenthesis unlike methods.



SYNTAX – METHOD CALLING

variable.method(...)

HELP: METHODS PER FUNCTION

help(function)

METHODS: STRINGS

Stringvar.capitalize()	first letter capitalized
Stringvar.replace(x, y)	replace all instances of x with y
Stringvar.index(x)	index of character in string
Stringvar.upper()	upper case
Stringvar.lower()	lower case

EXAMPLE – UPPER CASE STRINGS

```
# string to experiment with: place
place = "poolhouse"

# Use upper() on place: place_up
place_up = place.upper()


# Print out place and place_up
print(place)
print(place_up)

# Print out the number of o's in place
print(place.count("o"))
```

METHODS: FLOATS

Floatvar.bit_length(x)
Floatvar.conjugate(x)

METHODS: LISTS (DOES NOT CHANGE LIST)

Listvar.index(x)	get index of item
Listvar.count(x)	count number of occurences

METHODS: LISTS (CHANGES LIST)

Listvar.append(x)	add x to end of list
Listvar.remove(x)	removes first index of x from a list
Listvar.reverse(x)	reverses list order

EXAMPLE – LIST COUNT AND INDEX METHODS

```
# Create list areas
areas = [11.25, 18.0, 20.0, 10.75, 9.50]

# Print out the index of the element 20.0
print(areas.index(20.0))

# Print out how often 9.50 appears in areas
print(areas.count(9.50))
```

METHOD: RANGE

This function can be used to define a range of numbers, but will need to be iterated in order to create a list of numbers.

range(start, end, step (optional))

By default, a range is its own data type in Python, so we use an iterative statement to append all the items into a list, or a simple iterative print to obtain all of the values.

e.g.

```
rng_1 = range(0,6,1)
print(type(rng_1))
print(rng_1)
list_2 = []
for i in rng_1:
    print(i)
    list_2.append(i)
print(list_2)
```

```
>>> %Run 'String working.py'

  <class 'range'>
  range(0, 6)
  0
  1
  2
  3
  4
  5
  [0, 1, 2, 3, 4, 5]
```

DICTIONARIES

An example of having to use lists in an inefficient way is shown below;

```
pop = [30.55, 2.77, 39.21]
countries = ["china", "australia", "usa"]
ind_aus = countries.index("australia")
print(ind_aus)
print(pop[ind_aus])
```

```
>>> %Run 'String working.py'
  1
  2.77
```

Dictionaries are an alternative method to keep values associated to one another in separate lists.

SYNTAX: DICTIONARY CREATION

Dictionaries are implied by encompassing a list in curly brackets/braces { }.
A dictionary is sorted by keys, with value associated to each key, as follows;

Dictionary = {“key1”:value1, “key2”:value2, “key3:value3}

E.g.

```
dictionary_1 = {"china":30.55, "australia":2.77, "usa":39.21}
```

SYNTAX: LOOKUP KEY VALUE

Dictionaries can be ‘searched’ for a key’s values as opposed to index retrieval as initially shown with lists.

Variable[“key”]
Will retrieve the values of ‘key’ from the dictionary ‘variable’.

E.g.

```
dictionary_1 = {"china":30.55, "australia":2.77, "usa":39.21}
print(dictionary_1["china"])
```

```
>>> %Run 'String working.py'
  30.55
```

SUB-LISTS AS KEY VALUES

One of the best ways to use a dictionary is to associate sub-lists instead of single key values.

E.g.

```
dictionary_1 = {"china":[30.55, 1], "australia":[2.77, 2], "usa":[39.21, 3]}
print(dictionary_1["china"])
```

```
>>> %Run 'String working.py'
  [30.55, 1]
```

METHOD: KEYS

The Keys method will summarise the keys within a dictionary object.

E.g.
dictionary_1 = {"china":[30.55, 1], "australia":[2.77, 2], "usa":[39.21, 3]}
print(dictionary_1.keys())

```
>>> %Run 'String working.py'
  dict_keys(['china', 'australia', 'usa'])
```

ADD/UPDATE KEYS AND VERIFYING EXISTING

To add a key to a dictionary, simply state an additional key, and its values in a formula.

Dictionary["new key"] = key value

To verify the presence of a key in a dictionary, we use an 'in' statement to yield a Boolean.

"key name" in dictionary

E.g.
dictionary_1 = {"china":[30.55, 1], "australia":[2.77, 2], "usa":[39.21, 3]}
print(dictionary_1.keys())

print("canada" in dictionary_1)

dictionary_1["canada"] = 20.25
print(dictionary_1)

print("canada" in dictionary_1)

```
>>> %Run 'String working.py'
  dict_keys(['china', 'australia', 'usa'])
  False
  {'china': [30.55, 1], 'australia': [2.77, 2], 'usa': [39.21, 3], 'canada': 20.25}
  True
```

REMOVE KEYS

To remove a key from a dictionary, we use a delete function, and similar syntax to adding a key.

Del(Dictionary[“key”]

E.g.
dictionary_1 = {"china":[30.55, 1], "australia":[2.77, 2], "usa":[39.21, 3]}
print(dictionary_1.keys())

del(dictionary_1["china"])
print(dictionary_1.keys())

```
>>> %Run 'String working.py'

  dict_keys(['china', 'australia', 'usa'])
  dict_keys(['australia', 'usa'])
```

DICTIONARIES VS LISTS

The following table gives a better of idea of the key differences, and when to use each;

List	Dictionary
Select, update and remove: [ ]	Select, update and remove: [ ]
Indexed by range of numbers	Indexed by unique keys
Collection of values order matters select entire subsets	Lookup table with unique keys

NESTED DICTIONARIES

Dictionaries can contain any item that is considered immutable, which includes other dictionaries!
Dictionaries can contain sub-dictionaries, and can be key chained (like index chaining).

Dictionary['sub-dictionary'][key]

E.g.

```
europe = { 'spain': { 'capital':'madrid', 'population':46.77 },
        'france': { 'capital':'paris', 'population':66.03 },
        'germany': { 'capital':'berlin', 'population':80.62 },
        'norway': { 'capital':'oslo', 'population':5.084 } }

print(europe['france']['capital'])

data = {'capital':'rome', 'population':59.83}
europe['italy']=data

print(europe)
```

```
>>> %Run 'String working.py'

  paris
  {'spain': {'capital': 'madrid', 'population': 46.77}, 'france': {'capital': 'paris', 'population'
  : 66.03}, 'germany': {'capital': 'berlin', 'population': 80.62}, 'norway': {'capital': 'oslo', 'p
  opulation': 5.084}, 'italy': {'capital': 'rome', 'population': 59.83}}
```

EXAMPLE: CIRCULAR LOGIC AND PI

```
# Definition of radius
r = 0.43

# Import the math package
import math

# Calculate C
C = 2 * math.pi * r

# Calculate A
A = math.pi * (r ** 2)

# Build printout
print("Circumference: " + str(C))
print("Area: " + str(A))
```

EXAMPLE: CIRCULAR LOGIC AND RADIANS

```
# Definition of radius
r = 192500

# Import radians function of math package
from math import radians

# Travel distance of Moon over 12 degrees. Store in dist.
dist = r * radians(12)

# Print out dist
print(dist)
```

PYTHON NUMPY ARRAYS

VERSION 0.2

(Last edited 4/9/19)

PACKAGES - NUMPY

Packages are extensions to Python function/method libraries.

USEFUL PACKAGES

Numpy	data science
Matplotlib	visualization of data
Scikit-learn	machine based learning
Math	mathematical functions

FUNCTION: IMPORT PACKAGE (TO USE IT)

import packagename
import packagename as alias

E.g import numpy as np

NUMPY ARRAYS

Arrays are lists which allow laced functions to elements (element wise)
Need to manage types separately however, array by default converts to strings.

numpy.array([1,2,3])

[1,2,3]+[1,2,3]

as a list	[1,2,3,1,2,3]
as an array	[2,4,6]

treat an array like a list for various index retrival

e.g. array[1]

FUNCTION: ARRAY TO BOOLEAN VIA CONDITIONS

array is a data type, so has methods

Operations can be applied across all items in an array (like a list.map).

array > 25
may yield array([False, True, False], dtype=bool)

METHOD: FILTER BY CONDITION

to filter by boolean mask, subset the conditioned
e.g. array[array > 25]

EXAMPLE: ARRAY TIMES A FLOAT

```
# height is available as a regular list

# Import numpy
import numpy as np

# Create a numpy array from height_in: np_height_in
np_height_in = np.array(height_in)

# Print out np_height_in
print(np_height_in)

# Convert np_height_in to m: np_height_m
np_height_m = 0.0254 * np_height_in

# Print np_height_m
print(np_height_m)
```

EXAMPLE: CROSS TWO ARRAYS (BMI)

```
# height and weight are available as regular lists

# Import numpy
import numpy as np

# Create array from height_in with metric units: np_height_m
np_height_m = np.array(height_in) * 0.0254

# Create array from weight_lb with metric units: np_weight_kg
np_weight_kg = np.array(weight_lb)*0.453592

# Calculate the BMI: bmi
bmi = np_weight_kg / (np_height_m ** 2)

# Print out bmi
print(bmi)
```

EXAMPLE: FILTER ONE ARRAY BY A BOOLEAN ARRAY

```
# height and weight are available as a regular lists

# Import numpy
import numpy as np

# Calculate the BMI: bmi
np_height_m = np.array(height_in) * 0.0254
np_weight_kg = np.array(weight_lb) * 0.453592
bmi = np_weight_kg / np_height_m ** 2

# Create the light array
light = bmi<21

# Print out light
print(light)

# Print out BMIs of all baseball players whose BMI is below 21
print(bmi[light])
```

EXAMPLE: PACKAGE ALIAS’

```
import numpy
arr1 = numpy.array([1,2,3])
print(arr1)
```

or using an alias

```
import numpy as np
arr1 = np.array([1,2,3])
print(arr1)
```

you can call on one function only also (but you lose context)

```
from numpy import array
arr1 = array([1,2,3])
print(arr1)
```

NUMP ARRAY SHAPES/DIMENSIONS

A single list forms a 1D array
Sublists form a 2D array (rows/columns like dynamo)

Arrays desire a homogenous data type, so will adopt a common type such as string if it must

METHOD: SHAPE OF ARRAY

variable.shape return rows/columns of array as a count
eg. var.shape may yield (2, 5)

WORKING AT SUB-LIST/SET LEVELS IN ARRAYS

variable[X][Y] where X is the row and Y is the columns
or
variable[X,Y] X is the sublist, Y is the index in the sublist

variable[:,1:3] all sublists/rows, index 1 and 2 from both
variable[1,:] second sublist only

TRANSPOSITION OF DATA

data is automatically transposed to rows when you extract a column

EXAMPLE: SUBLIST RETRIEVAL FROM ARRAYS

```
# baseball is available as a regular list of lists

# Import numpy package
import numpy as np

# Create np_baseball (2 cols)
np_baseball = np.array(baseball)

# Print out the 50th row of np_baseball
print(np_baseball[50,:])

# Select the entire second column of np_baseball: np_weight_lb
np_weight_lb = np_baseball[:,1]

# Print out height of 124th player
np_height_lb = np_baseball[:,0]
print(np_height_lb[123])
```

APPLYING FUNCTIONS ACROSS ARRAYS

Numpy arrays can be multiplied and added, if it's a single row it will multiply for each list

example

```
import numpy as np
np_mat = np.array([[1, 1],
                   [2, 2],
                  [3, 3]])
print(np_mat * 2)
print(np_mat + np.array([1, 2]))
print(np_mat + np_mat)
```

Output

```
[[2 2]
 [4 4]
 [6 6]]
[[2 3]
 [3 4]
 [4 5]]
[[2 2]
 [4 4]
 [6 6]]
```

METHODS: ARRAYS, VARIOUS

np.mean(array)	average value
np.median(array)	actual middle value
np.corrcoef(array1, array2)	correlation between X and Y as a coefficient
np.std(array)	standard deviation (level of variance either side of mean that is reasonable
np.round(X, Y)	round X to Y sig figs.
np.random.normal(X, Y, Z)	random range, mean/standard dev/samples to make
np.column_stack	sub-transpose a list into two sublists
np.transpose(X)	transpose the array structure X

EXAMPLE: MEAN MEDIAN OF AN ARRAY

```
# np_baseball is available

# Import numpy
import numpy as np

# Create np_height_in from np_baseball
np_height_in = np_baseball[:,0]
print(np_height_in)

# Print out the mean of np_height_in
print(np.mean(np_height_in))

# Print out the median of np_height_in
print(np.median(np_height_in))
```

EXAMPLE: CORRELATION AND STANDARD DEVIATION

correlation is between -1 and 1, 1 being strong, -1 being strongly non-related, 0 means little relationship

```
# np_baseball is available

# Import numpy
import numpy as np

# Print mean height (first column)
avg = np.mean(np_baseball[:,0])
print("Average: " + str(avg))

# Print median height. Replace 'None'
med = np.median(np_baseball[:,0])
print("Median: " + str(med))

# Print out the standard deviation on height. Replace 'None'
stddev = np.std(np_baseball[:,0])
print("Standard Deviation: " + str(stddev))

# Print out correlation between first and second column. Replace 'None'
corr = np.corrcoef(np_baseball[:,0], np_baseball[:,1])
print("Correlation: " + str(corr))
```

EXAMPLE: MEAN MEDIAN OF AN ARRAY DERIVED FROM LISTS

```
# heights and positions are available as lists

# Import numpy
import numpy as np

# Convert positions and heights to numpy arrays: np_positions, np_heights
np_positions = np.array(positions)
np_heights = np.array(heights)

# Heights of the goalkeepers: gk_heights
gk_heights = np_heights[np_positions == 'GK']

# Heights of the other players: other_heights
other_heights = np_heights[np_positions != 'GK']

# Print out the median height of goalkeepers. Replace 'None'
print("Median height of goalkeepers: " + str(np.median(gk_heights)))

# Print out the median height of other players. Replace 'None'
print("Median height of other players: "+ str(np.median(other_heights)))
```

FUNCTION: RANDOM NUMBER GENERATION

Numpy has a package that can generate random outcomes. This is important to simulate chance based simulation.

Np.random.seed(value)
Will set a seeding value for random functions. It is good to know your seed so we can compare studies with less bias.

Np.random.rand()
Will provide a random number between 0 and 1.

Np.random.randint(X, Y)
Will provide a random integer from X up to Y (non-inclusive of Y)

E.g.

```
import numpy as np
np.random.seed(123)
tosses = range(15)
for coin in tosses:
   coin = np.random.randint(0,2)
   if coin == 0:
      print("heads")
   else:
      print("tails")
```

```
>>> %Run rand.py

  heads
  tails
  heads
  heads
  heads
  heads
  heads
  tails
  tails
  heads
  tails
  tails
  heads
  tails
  heads
```

CHANCE BASED LOOPS

We can use a ‘while’ loop to conduct a study until a condition is met.
In the below example, we toss a coin until we throw 50 heads.

E.g.

```
import numpy as np
np.random.seed(123)
tosses = 0
total = 0
heads = 0
tails = 0
while total < 50:
    coin = np.random.randint(0,2)
    tosses += 1
    if coin == 0:
        total += 1
        heads += 1
    else:
        tails += 1

print("TOTAL: "+str(tosses))
print("HEADS: "+str(heads))
print("TAILS: "+str(tails))
```

```
>>> %Run rand.py

  TOTAL: 87
  HEADS: 50
  TAILS: 37
```

RANDOM WALKS

A random walk will track the number of outcomes in the list at that point, and is a better way of tracking when something occurs. We can use a simple 'for' loop to generate this:

E.g.

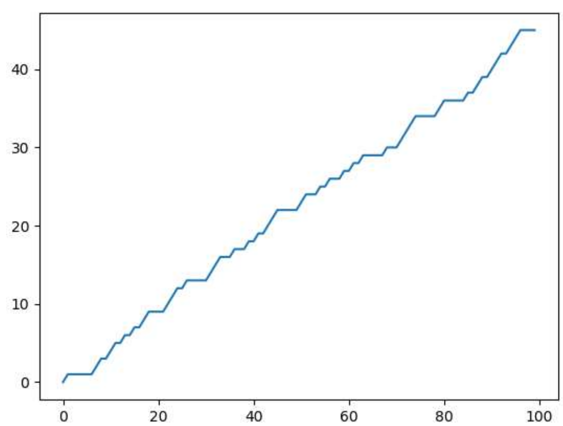
```
import numpy as np
np.random.seed(123)
tails = [0]
for x in range(10):
    coin = np.random.randint(0,2)
    tails.append(tails[x] + coin)
print(tails[1:])
```

```
>>> %Run rand.py
  [0, 1, 1, 1, 1, 1, 1, 2, 3, 3]
```

This reflects our previous results, but with each step tracked individually.

```
>>> %Run rand.py
  ['heads', 'tails', 'heads', 'heads', 'heads', 'heads', 'heads', 'tails', 'tails', 'heads']
```

Don't forget we can use matplotlib.pyplot to visually show walk based progressions:



MULTIPLE RUNS

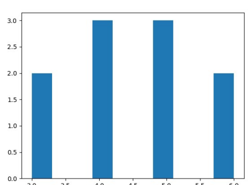
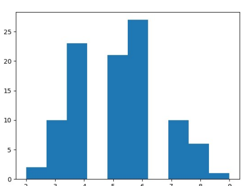
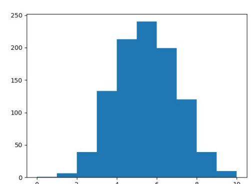
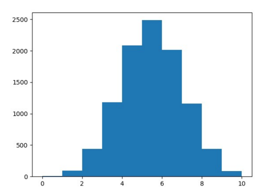
We can nest a loop within another in order to run multiple runs of multiple outcomes.

We can also use a histogram to plot the distribution of the results. In the case below we are tossing a coin, so if we increase our runs, we will get closer to a bell curve distribution once we increase the studies. The below results are for 10, 100, 1000 and 10000 tosses respectively.

E.g.

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(123)
final_tails = []
for x in range(10000):
   tails = [0]
   for x in range(10):
      coin = np.random.randint(0,2)
      tails.append(tails[x] + coin)
   final_tails.append(tails[-1])
print(final_tails)
plt.hist(final_tails, bins = 10)
plt.show()
```

```
>>> %Run rand.py
 [3, 6, 4, 5, 4, 5, 3, 5, 4, 6, 6, 8, 6, 4, 7, 5, 7, 4, 3, 3, 4, 5, 8, 5, 6, 5, 7, 6, 4, 5, 8, 5,
 8, 4, 6, 6, 3, 4, 5, 4, 7, 8, 9, 4, 3, 4, 5, 6, 4, 2, 6, 6, 5, 7, 5, 4, 5, 5, 6, 7, 6, 6, 6, 3, 6
 , 3, 6, 5, 6, 5, 6, 4, 6, 6, 3, 4, 4, 2, 4, 5, 4, 6, 6, 6, 8, 4, 6, 5, 7, 4, 6, 5, 4, 6, 7, 3, 7,
  4, 5, 7]
```

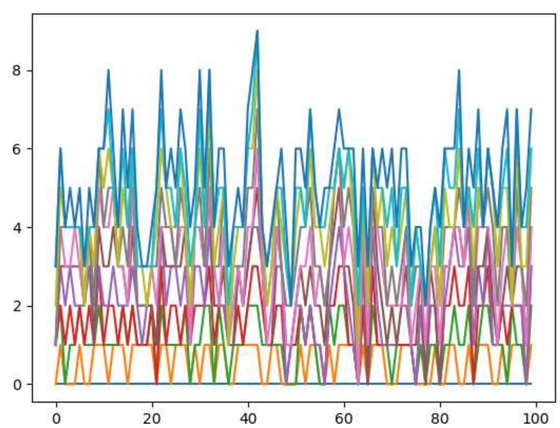
MULTIPLE RUNS AS AN ARRAY

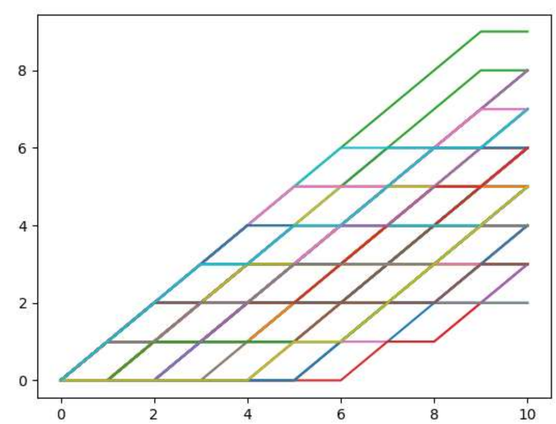
Converting our results to a Numpy array before visualising will show us the average trends over our results. If we transpose our data, we can chart the linear progression of each random walk on a sequential scale.

In the case below, the left graph is an array, and the one adjacent is transposed.

E.g.

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(123)
final_tails = []
for x in range(100):
   tails = [0]
   for x in range(10):
      coin = np.random.randint(0,2)
      tails.append(tails[x] + coin)
   final_tails.append(tails)
np_ft = np.array(final_tails)
np_ft_t = np.transpose(np_ft)
plt.plot(np_ft_t)
plt.show()
```





Pandas

PYTHON FOR SPREADSHEET USERS

VERSION 0.2

(Last edited 4/9/19)

PACKAGE – PANDAS/XLRD

Used for statistics, data analysis and time. Typically works with rows/columns from Excel.
Variables may be constantly redefined in a specific order if required.

You may also need to import XLRD as well for reading Excel data.

Commonly imported as 'pd' for short.
Import pandas as pd

FUNCTION: READ EXCEL

Set your document as a file path in a separate variable first.

```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
print(df)
```

METHOD: HEAD

Limits the amount of data previewed to a set number.

```
variable = pd.read_excel (doc_1)
doc_1.head(X)
```

X is the number of indexed rows to show from the start.
By default, X is 5 if not specified.

METHOD: INFO

This Method displays information about the table such as variable types and data counts.

```
doc_1.info
```

E.g.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 3 columns):
header 1    32 non-null int64
header 2    32 non-null int64
header 3    32 non-null int64
dtypes: int64(3)
memory usage: 808.0 bytes
None
```

Object	= Text
Int64	= Integer/whole
Float64	= Numerical data

METHOD: DESCRIBE

This Method displays information about the data when it is numerically reviewable

doc1_describe

E.g.

	header 1	header 2	header 4
count	26.000000	26.000000	26.000000
mean	2.423077	6.423077	6.423077
std	1.137474	1.137474	1.137474
min	1.000000	5.000000	5.000000
25%	1.250000	5.250000	5.250000
50%	2.000000	6.000000	6.000000
75%	3.000000	7.000000	7.000000
max	4.000000	8.000000	8.000000

METHOD: SORT VALUES

This Method reorganises a table based on a column in the table.

doc1.sort_values('X')

X is the header text in the column to be sorted by.

You may optionally set whether the order is ascending after X;
doc1_sort_values('X', **ascending=False**)

All other columns will be sorted. Note that indexes are not reset.

E.g.
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
df_sort1 = df.sort_values(header, ascending=True)
print(df_sort1)

	header 1	header 2	header 3	header 4
0	1	5	b	5
1	2	6	r	6
2	3	7	t	7
3	4	8	s	8
4	1	5	d	5
5	2	6	f	6
6	3	7	c	7
7	4	8	q	8
8	1	5	r	5
9	2	6	y	6
10	3	7	i	7
11	4	8	f	8
12	1	5	q	5
13	2	6	d	6
14	3	7	h	7

	header 1	header 2	header 3	header 4
19	4	8	a	8
0	1	5	b	5
6	3	7	c	7
25	2	6	c	6
4	1	5	d	5
18	3	7	d	7
13	2	6	d	6
5	2	6	f	6
11	4	8	f	8
14	3	7	h	7
10	3	7	i	7
17	2	6	j	6
15	4	8	k	8
21	2	6	k	6
22	3	7	l	7

METHOD: RESET INDEX

This Method resets all indices of rows to top to bottom order, useful for resetting sorted indices.

doc1_reset_index(**drop=True**)

Note that if drop is set to False, a second column will be introduced after the index column.
This column contains the old index values prior to being sorted.

E.g.
```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
df_sort1 = df.sort_values(header, ascending=True)
df_sort1R = df_sort1.reset_index(drop=False)
print(df_sort1R)
```

	index	header 1	header 2	header 3	header 4
0	19	4	8	a	8
1	0	1	5	b	5
2	6	3	7	c	7
3	25	2	6	c	6
4	4	1	5	d	5
5	18	3	7	d	7
6	13	2	6	d	6
7	5	2	6	f	6
8	11	4	8	f	8
9	14	3	7	h	7
10	10	3	7	i	7
11	17	2	6	j	6

ISOLATE ROWS

Variable[X:Y]

The above will return the rows at these indices, not including the header row.

ISOLATE COLUMNS, RUN LOGIC ACROSS COLUMNS

Variable['X']

The above will return the column in the table with header title X only.
The data is still structured with index numbers, as well as information about the column below.

If we use double square brackets, the indices will also be retained, and a dataframe will result as opposed to a series.

E.g.

```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
df_sort1 = df.sort_values(header, ascending=True)
df_sort1R = df_sort1.reset_index(drop=False)
print(df_sort1R[header])
```

We can also apply logic checks (==, !=, <, >, <=, >=) to the data;

E.g.

```
print(df_sort1R[header] == y)
```

```
0     a
1     b
2     c
3     c
4     d
5     d
6     d
7     f
8     f
9     h
10    i
11    j
12    k
13    k
14    l
15    p
16    q
17    q
18    r
19    r
20    s
21    s
22    t
23    t
24    y
25    y
Name: header 3, dtype: object
```

```
0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12    False
13    False
14    False
15    False
16    False
17    False
18    False
19    False
20    False
21    False
22    False
23    False
24     True
25     True
Name: header 3, dtype: bool
```

FILTER ROWS BY COLUMN

We can then nest our data isolation and logic check one level further in order to filter the data.

E.g.
Following on from previous;
print(df_sort1R[df_sort1R[header] == 'y'])

```
    index  header 1  header 2 header 3  header 4
24     24         1         5        y         5
25      9         2         6        y         6
```

You can add a reset_index after these types of functions in order to reset indexes immediately.

E.g.
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
df_sort1 = df.sort_values(header, ascending=True)
df_sort1R = df_sort1.reset_index(drop=True)
print(df_sort1R[df_sort1R[header] == 'y'].reset_index(drop=True))

```
    index  header 1  header 2 header 3  header 4
24     24         1         5        y         5
25      9         2         6        y         6
```

Becomes

```
   index  header 1  header 2 header 3  header 4
0     24         1         5        y         5
1     25         2         6        y         6
```

ADDING NEW COLUMNS

We can introduce columns at the end of data using mathematical operators.
All we need to do is define a new column as a variable, and it will be included in the table when next called upon.

E.g.

```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
header1 = 'header 4'
df = df.sort_values(header, ascending=True)
df = df.head(3)
df['new_col'] = df['header 4'] * 2
print(df)
```

```
    header 1  header 2 header 3  header 4  new_col
19         4         8        a         8       16
0          1         5        b         5       10
6          3         7        c         7       14
```

We can also cross reference multiple columns in arithmetic style statements.

E.g.

```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
header = 'header 3'
df = df.sort_values(header, ascending=True)
df = df.head(3)
df['new_col'] = df['header 4'] * df['header 2']
print(df)
```

```
    header 1  header 2 header 3  header 4  new_col
19         4         8        a         8       64
0          1         5        b         5       25
6          3         7        c         7       49
```

METHOD: COLUMN SUM

We can use the sum function to get totals out of columns.
If we do not use numeric only, we will get string concatenations.

E.g.
```
import pandas as pd
doc_1 = r'C:\Users\GavinC\Desktop\doc1.xlsx'
df = pd.read_excel (doc_1)
dfsums = df.sum(numeric_only=True)
print(dfsums)
```

```
>>> %Run spreadsheets.py

  header 1    13
  header 2    37
  header 4    37
  dtype: int64
```

METHOD: GROUPBY (PIVOT) – 'METHOD CHAINING'

We can use grouping based on a column with repeated values to pivot our data.
Any functions following the groupby will apply to the data. Sum is an example of just one method available.

This is called method chaining.

If index = True, the pivot handle will become the index instead of actual numbers.

E.g.
```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
df = pd.read_excel(doc_1)
dfpivot_1 = df.groupby("store",as_index=False).sum()
print(dfpivot_1)
```

```
>>> %Run Pandas.py

                   store product_name  quantity_purchased  revenue
  0   Pete's Discount Fruit       Banana                   1     0.23
  1     Derek's Fruit Stand       Banana                   3     0.69
  2   Pete's Discount Fruit       Orange                   1     0.68
  3     Derek's Fruit Stand       Orange                   2     1.36
  4   Pete's Discount Fruit        Apple                   1     0.88
  5     Derek's Fruit Stand        Apple                   1     0.88
```

```
>>> %Run Pandas.py

                   store  quantity_purchased  revenue
  0    Derek's Fruit Stand                  16    28.07
  1  Pete's Discount Fruit                  12    22.06
```

METHOD: GROUPBY MULTIPLE (PIVOT) – 'METHOD CHAINING'

E.g.
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
df = pd.read_excel(doc_1)
groups_1 = ['store', 'product_name']
dfgrouped_1 = df.groupby(groups_1, as_index=False).sum()
print(dfgrouped_1)

```
>>> %Run Pandas.py

                    store product_name  quantity_purchased  revenue
0      Derek's Fruit Stand        Apple                   1     0.88
1      Derek's Fruit Stand       Banana                   4     0.92
2      Derek's Fruit Stand  Dragonfruit                   3    15.81
3      Derek's Fruit Stand         Kiwi                   2     2.24
4      Derek's Fruit Stand       Orange                   2     1.36
5      Derek's Fruit Stand         Plum                   3     2.88
6      Derek's Fruit Stand   Watermelon                   1     3.98
7    Pete's Discount Fruit        Apple                   4     3.52
8    Pete's Discount Fruit       Banana                   2     0.46
9    Pete's Discount Fruit  Blueberries                   3    15.48
10   Pete's Discount Fruit       Orange                   1     0.68
11   Pete's Discount Fruit         Plum                   2     1.92
```

Data camp example (combining functions)

# Summarize by movie title and ticket type
movies_by_ticket_type = sales.groupby(['movie_title', 'ticket_type'], as_index=False).sum()

# Filter for senior tickets
senior_ticket_movies = movies_by_ticket_type[movies_by_ticket_type['ticket_type'] == 'senior'].reset_index(drop=True)

# Sort senior ticket sales descending
ordered_senior_movies = senior_ticket_movies.sort_values('ticket_quantity', ascending=False).reset_index(drop=True)

# Print the top 3 rows of the ordered table
print(ordered_senior_movies.head(3))

METHOD: HEAD – TOP ROW PER PIVOTED ITEM

We can obtain the top X results of a sorted list in order to compare trends more easily.

E.g.

```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
df = pd.read_excel(doc_1)
groups_1 = ['store', 'product_name']
dfgrouped_1 = df.groupby(groups_1, as_index=False).sum()
dfsorted_1 = dfgrouped_1.sort_values('quantity_purchased', ascending=False)
dfpopular_1 = dfsorted_1.groupby('store').head(2).reset_index(drop=True)
print(dfpopular_1)
```

```
>>> %Run Pandas.py

                    store product_name  quantity_purchased  revenue
 1      Derek's Fruit Stand       Banana                   4     0.92
 7    Pete's Discount Fruit        Apple                   4     3.52
 2      Derek's Fruit Stand  Dragonfruit                   3    15.81
 5      Derek's Fruit Stand         Plum                   3     2.88
 9    Pete's Discount Fruit  Blueberries                   3    15.48
 3      Derek's Fruit Stand         Kiwi                   2     2.24
 4      Derek's Fruit Stand       Orange                   2     1.36
 8    Pete's Discount Fruit       Banana                   2     0.46
 11   Pete's Discount Fruit         Plum                   2     1.92
 0      Derek's Fruit Stand        Apple                   1     0.88
 6      Derek's Fruit Stand   Watermelon                   1     3.98
 10   Pete's Discount Fruit       Orange                   1     0.68
```

```
>>> %Run Pandas.py

                    store product_name  quantity_purchased  revenue
 0      Derek's Fruit Stand       Banana                   4     0.92
 1    Pete's Discount Fruit        Apple                   4     3.52
 2      Derek's Fruit Stand  Dragonfruit                   3    15.81
 3    Pete's Discount Fruit  Blueberries                   3    15.48
```

FUNCTION: READ MULTIPLE EXCEL TABS

This Function is for workbooks with multiple tabs instead of just one to be read.

pd.ExcelFile('variable')

Note that initially, this is not a data frame that can be printed.

E.g.
```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
df = pd.ExcelFile(doc_1)
print(df)
```

```
>>> %Run Pandas.py
  <pandas.io.excel._base.ExcelFile object at 0x0982BFB0>
```

From here, we access the Attributes of the Excel object.

E.g.
```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
df = pd.ExcelFile(doc_1)
df_sheets = df.sheet_names
print(df_sheets)
```

```
>>> %Run Pandas.py
  ['Sheet1', 'Price', 'Color']
```

METHOD: PARSE

A multi-tab workbook can be accessed using this method.
Note that the sheet name attributes list can also be fed into this method for all data in a workbook.

Workbook.parse('sheet name')

E.g.
```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)
sheets_1 = workbook_1.sheet_names
prices_1 = workbook_1.parse('Price')
print(prices_1)
```

```
>>> %Run Pandas.py
            name  price_usd
  0        Apple       0.88
  1       Banana       0.23
  2       Orange       0.68
  3   Watermelon       3.98
  4         Plum       0.96
  5  Blueberries       5.16
  6  Dragonfruit       5.27
  7         Kiwi       1.12
```

DATA COMPATIBILITY FOR MERGING

Important things to note when merging Python data;

- Python is case-sensitive!
  - `'Apple' != 'apple'`
- Python is exact!
  - `'Apple' != ' Apple '`
- Python joins whole tables!
  - AB + ACD = ABCD, even if we only want ABC
  - So we have to drop column D.

METHOD: STR.XXX

Case must match between data sets. Typically, it is best to set up data filters on certain columns to ensure case is consistent.

Variable['header'] = Variable['header'].str.upper()
Variable['header'] = Variable['header'].str.title()
Variable['header'] = Variable['header'].str.lower()

To remove padded white space on the left or right, we use the strip function.

Variable['header'] = Variable['header'].str.strip()

```
E.g.
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)
sheets_1 = workbook_1.sheet_names
prices_1 = workbook_1.parse('Price')
prices_1['name'] = prices_1['name'].str.upper()
print(prices_1)
```

```
>>> %Run Pandas.py

          name  price_usd
0        APPLE       0.88
1       BANANA       0.23
2       ORANGE       0.68
3   WATERMELON       3.98
4         PLUM       0.96
5  BLUEBERRIES       5.16
6  DRAGONFRUIT       5.27
7         KIWI       1.12
```

METHOD: DROP COLUMNS

To remove redundant columns, feed in a single or list of the column header names to be dropped.

Variable[‘header’] = Variable.drop(‘header’’, axis=1)

E.g.

```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)
sheets_1 = workbook_1.sheet_names
prices_1 = workbook_1.parse('Price')
prices_1['name'] = prices_1['name'].str.upper()
prices_1 = prices_1.drop('price_usd', axis=1)
print(prices_1)
```

```
>>> %Run Pandas.py

          name  price_usd
0        APPLE       0.88
1       BANANA       0.23
2       ORANGE       0.68
3   WATERMELON       3.98
4         PLUM       0.96
5  BLUEBERRIES       5.16
6  DRAGONFRUIT       5.27
7         KIWI       1.12

>>> %Run Pandas.py

          name
0        APPLE
1       BANANA
2       ORANGE
3   WATERMELON
4         PLUM
5  BLUEBERRIES
6  DRAGONFRUIT
7         KIWI
```

METHOD: MERGING DATA (VLOOKUP OF PYTHON)

Similarly, to excel, we can work across mulontiple data sets to combine tables.

Key columns are used to match the data sets between the two.
We do what is called joining to merge data. Left is if we are attaching data from a later tab, right for a prior tab.

Python will return an N/A if a lookup value is not present for merging.

Table1.merge(table2, on = ‘key header’, how = ‘left’)

Or if headers are not the same name, we can specify the name in each tab

Table1.merge(table2, left_on = ‘key header 1’, right_on = ‘key header 2’, how = ‘left’)

E.g.

```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)
sheets_1 = workbook_1.sheet_names
prices_1 = workbook_1.parse('Price')
colors_1 = workbook_1.parse('Color')
combined_1 = colors_1.merge(prices_1, on='name', how='left')
print(combined_1)
```

```
>>> %Run Pandas.py
          name   color  price_usd
0        Apple     red       0.88
1       Banana  yellow       0.23
2       Orange  orange       0.68
3   Watermelon   green       3.98
4         Plum  purple       0.96
5  Blueberries    blue       5.16
6  Dragonfruit    pink       5.27
7         Kiwi   brown       1.12
```

DICTIONARY TO PANDAS (DATAFRAME)

Dictionaries are a great method to establish a dataframe (table) of objects for a package like pandas to work with.
The key of each row should be your column label, and the values are that key's values respectively.

Dataframe = pd.DataFrame(dictionary)
Will convert your dictionary to a data frame structure, able to be manipulated by pandas.

E.g.

```
import pandas as pd

dict = {
   "country":["Brazil", "Russia", "India", "China", "South Africa"],
   "capital":["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"],
   "area":[8.516, 17.10, 3.286, 9.597, 1.221],
   "population":[200.4, 143.5, 1252, 1357, 52.98]}

world = pd.DataFrame(dict)
```

```
>>> %Run 'String working.py'

        country    capital    area  population
0        Brazil   Brasilia   8.516      200.40
1        Russia     Moscow  17.100      143.50
2         India  New Delhi   3.286     1252.00
3         China    Beijing   9.597     1357.00
4  South Africa   Pretoria   1.221       52.98
```

DATAFRAME INDEX PROCESSING

By default, a pandas data frame will use numbers for index references. A more efficient method would be to process a key list from the dictionary, and manipulate its data to form more meaningful keys.

The example below uses a looping statement in order to process each country name into a list of index values.

E.g.

```
import pandas as pd

dict = {
   "country":["Brazil", "Russia", "India", "China", "South Africa"],
   "capital":["Brasilia", "Moscow", "New Delhi", "Beijing", "Pretoria"],
   "area":[8.516, 17.10, 3.286, 9.597, 1.221],
   "population":[200.4, 143.5, 1252, 1357, 52.98]}

dkeys_l = dict["country"]
print(dkeys_1)

dkeys_s = []
for i in dkeys_l:
   i = i[:2]
   dkeys_s.append(i.upper())
print(dkeys_s)

world = pd.DataFrame(dict)
world.index = dkeys_s

print(world)
```

```
>>> %Run 'String working.py'

  ['BR', 'RU', 'IN', 'CH', 'SO']
           country    capital    area  population
  BR        Brazil   Brasilia   8.516      200.40
  RU        Russia     Moscow  17.100      143.50
  IN         India  New Delhi   3.286     1252.00
  CH         China    Beijing   9.597     1357.00
  SO  South Africa   Pretoria   1.221       52.98
```

DATAFRAME FROM CSV FILE

It is rare for us to write an entire dataframe structure within Python as previous examples have done.
We can interpret a CSV (comma separated value) file using pandas in order to process data to a dataframe easily.

Dataframe = pd.read_csv("file path", index_col = N)
If you do not specify an index column, your dataframe will be given a numerical index in addition.

E.g.
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)

countries.csv - Notepad

File Edit Format View Help

```
,country,capital,area,population
BR,Brazil,Brasilia,8.516,200.4
RU,Russia,Moscow,17.10,143.5
IN,India,New Delhi,3.286,1252
CH,China,Beijing,9.597,1357
SA,South
Africa,Pretoria,1.221,52.98
```

```
>>> %Run CSV.py

          country    capital    area  population
BR         Brazil   Brasilia   8.516      200.40
RU         Russia     Moscow  17.100      143.50
IN          India  New Delhi   3.286     1252.00
CH          China    Beijing   9.597     1357.00
SA   South Africa   Pretoria   1.221       52.98
```

METHOD: LOC

This method allows us to find rows of data based on matching indices.

Dataframe.loc[“Index”]

```
E.g.
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)

loc_1 = data_1.loc["RU"]
print()
print(loc_1)
```

```
>>> %Run CSV.py

         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98

country       Russia
capital       Moscow
area            17.1
population     143.5
Name: RU, dtype: object
```

We can obtain the data frame structure by encapsulating with an extra pair of square brackets.

```
E.g.
…
loc_1 = data_1.loc[["RU"]]
…
```

```
>>> %Run CSV.py

         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98

   country capital  area  population
RU  Russia  Moscow  17.1       143.5
```

We can obtain multiple rows, as well as isolated columns using the following syntax:

Dataframe.loc[[“Index1”, “Index2”, “Index3”], [“Header1”, “Header2”]]

E.g.
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)

loc_1 = data_1.loc[["RU", "IN", "CH"], ["country", "capital"]]
print()
print(loc_1)

```
>>> %Run CSV.py

          country    capital    area  population
BR         Brazil   Brasilia   8.516      200.40
RU         Russia     Moscow  17.100      143.50
IN          India  New Delhi   3.286     1252.00
CH          China    Beijing   9.597     1357.00
SA   South Africa   Pretoria   1.221       52.98

   country    capital
RU  Russia     Moscow
IN   India  New Delhi
CH   China    Beijing
```

If you replace the list of keys with a semicolon, you will get all rows, but only specified columns:

Dataframe.loc[:, [“Header1”, “Header2”]]

E.g.
…
loc_1 = data_1.loc[:, ["country", "capital"]]
…

```
>>> %Run CSV.py

          country    capital    area  population
BR         Brazil   Brasilia   8.516      200.40
RU         Russia     Moscow  17.100      143.50
IN          India  New Delhi   3.286     1252.00
CH          China    Beijing   9.597     1357.00
SA   South Africa   Pretoria   1.221       52.98

          country    capital
BR         Brazil   Brasilia
RU         Russia     Moscow
IN          India  New Delhi
CH          China    Beijing
SA   South Africa   Pretoria
```

METHOD: ILOC

This method allows us to find rows of data based on specified numerical indices.
This is almost the same as the Loc function, but we are using numbers vs. values.

		0	1	2	3
		country	capital	area	population
0	BR	Brazil	Brasilia	8.516	200.40
1	RU	Russia	Moscow	17.100	143.50
2	IN	India	New Delhi	3.286	1252.00
3	CH	China	Beijing	9.597	1357.00
4	SA	South Africa	Pretoria	1.221	52.98

Dataframe.loc["Index"]

E.g.

```
import pandas as pd
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Countries.csv'
data_1 = pd.read_csv(doc_1, index_col = 0)
print(data_1)

loc_1 = data_1.iloc[:, [0, 1]]
print()
print(loc_1)
```

```
>>> %Run CSV.py

         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98

         country    capital
BR        Brazil   Brasilia
RU        Russia     Moscow
IN         India  New Delhi
CH         China    Beijing
SA  South Africa   Pretoria
```

PYTHON DATA VISUALISATION

VERSION 0.2

(Last edited 4/9/19)

PACKAGE – MATPLOTLIB/SEABORN

Usually imported as these alias';

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

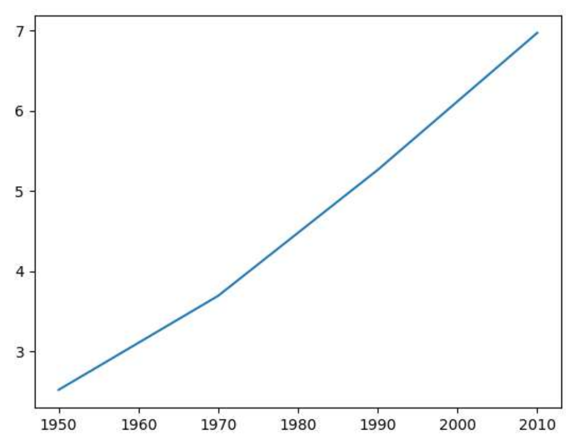
Seaborn is a package that makes matplotlib easier to use.
Matplotlib generates and displays graphs.

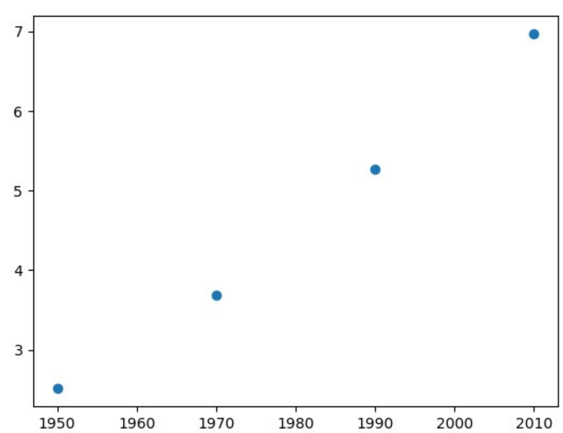
PLOTTING A LINE/SCATTER FUNCTION

A line function is one of the most basic plots using matplotlib. Scatters are also very common methods for visualising two dimensional data sets.

plt.plot(x,y)
or
plt.scatterplot(x,y)

E.g.
import matplotlib.pyplot as plt
list_1 = [1950, 1970, 1990, 2010]
list_2 = [2.519, 3.692, 5.263, 6.972]
plt.plot(list_1,list_2)
or
plt.scatter(list_1,list_2)
plt.show()





SCATTER CUSTOMISATION OPTIONS

Scattered data can be hard to interpet. We can also add variables such as scale and colour to our scattered data in order to more easily discern other trends and visualise our data in additional dimensions.

plt.scatter, x = list1, y = list2, s = list3, c = list4, alpha = number)
Where s represents a list of sizes to scale by, and c represents a list of colours to apply to our dots.
Alpha can be between 0 (transparent) and 1 (opaque).

plt.text(x, y, 'text')
Will add text labels on the graph at the x/y coordinate specified

plt.grid(True)
Will draw grid lines at all labels shown or specified.

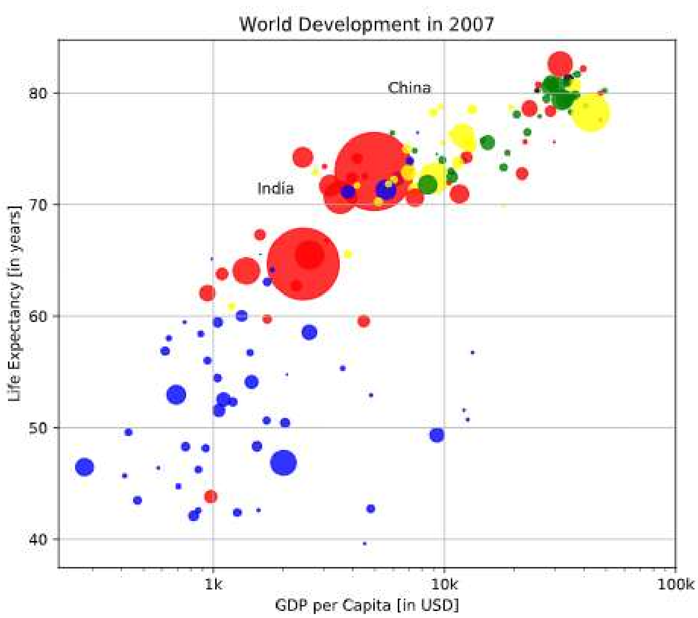
E.g.

```
plt.scatter(x = gdp_cap, y = life_exp, s = np.array(pop) * 2, c = col, alpha = 0.8)

plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')
plt.xticks([1000,10000,100000], ['1k','10k','100k'])

plt.text(1550, 71, 'India')
plt.text(5700, 80, 'China')
plt.grid(True)

plt.show()
```

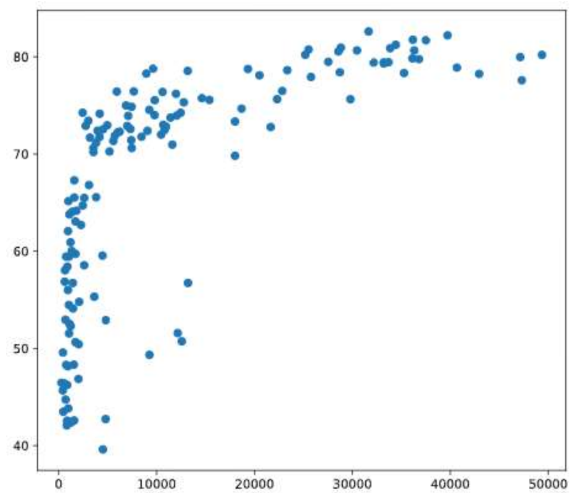


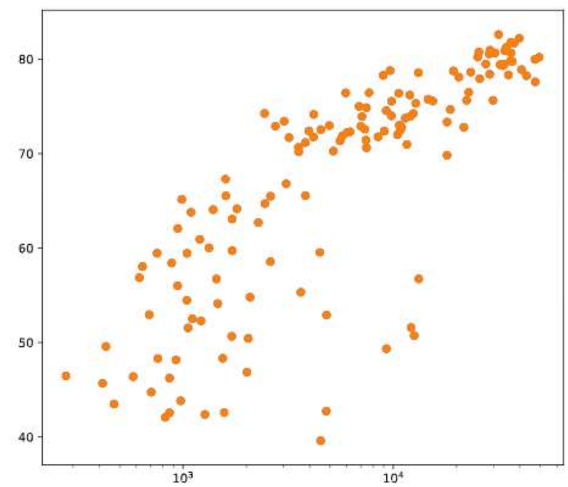
X/Y AXIS SCALING

Often it is easier to identify trends and correlations in scatter plots by applying a logarithmic scale to either an X or Y axis. This method of scaling will adjust the spacing between values, so that as they go further along, they are less spaced out to cover more distance.

plt.xscale('log')
or
plt.yscale('log')

E.g.
```
plt.scatter(gdp_cap, life_exp)
plt.xscale('log')
plt.show()
```





CLEANING UP A PLOT

Multiple plots can be generated from Python, but they must be cleaned first.

plt.clf()

E.g.
```
import matplotlib.pyplot as plt
list_1 = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]
plt.hist(list_1, bins=3)
plt.show()
plt.clf()
plt.hist(list_1, bins=2)
plt.show()
```

AXIS LABELS, TITLES AND TICKS

After we have made a plot, but before we show it, we can modify certain aspects of the presentation.

plt.title('title')
Changes the title of the graph

plt.xlabel('title')
plt.ylabel('title')
Changes the title of the axis labels

The axis labels are referred to as 'ticks'. We can modify them using certain methods.

Plt.yticks(list1, list2)
Will change the values (list1) and labels (list2) shown on the X or Y axis.

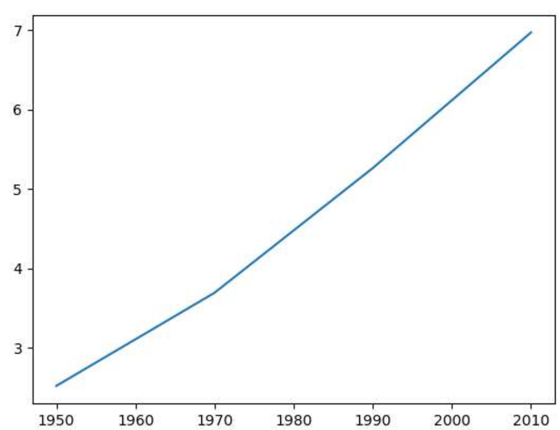
E.g.

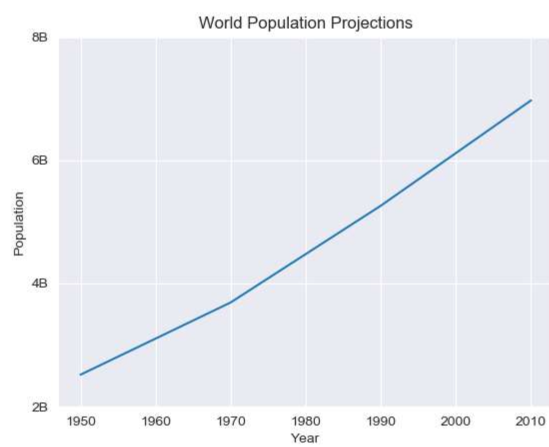
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')

list_1 = [1950, 1970, 1990, 2010]
list_2 = [2.519, 3.692, 5.263, 6.972]
plt.plot(list_1,list_2)

plt.xlabel('Year')
plt.ylabel('Population')
plt.title('World Population Projections')
plt.yticks([2,4,6,8],
        ['2B','4B','6B','8B'])

sns.despine()
plt.show()
```





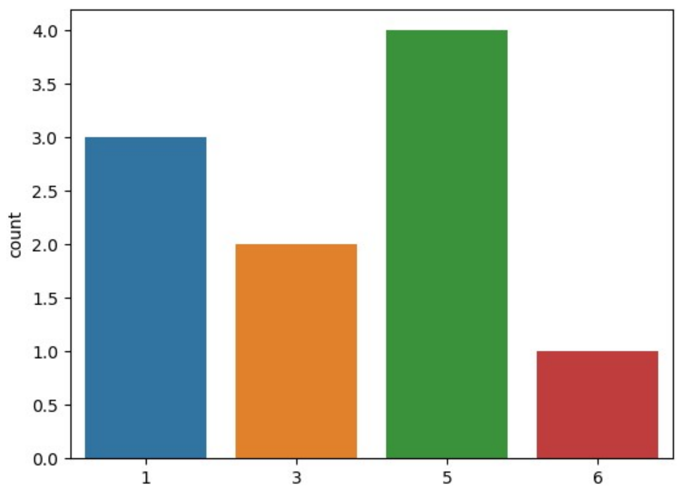
PLOTTING A LIST COUNT

List counts can be plotted using both matplotlib and seaborn.

s.scountplot(list)

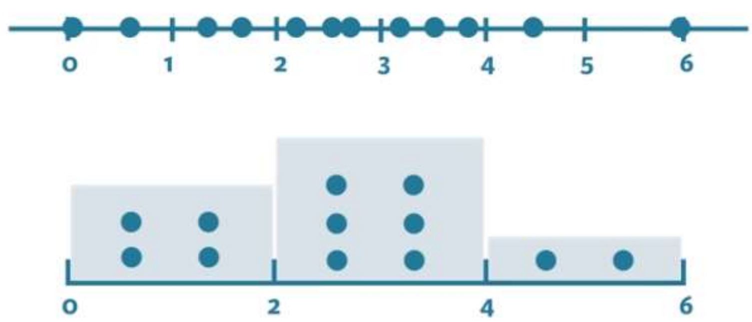
E.g.

```
import matplotlib.pyplot as plt
import seaborn as sns
list_1 = [1, 1, 1, 3, 3, 5, 5, 5, 5, 6]
sns.countplot(list_1)
plt.show()
```



PLOTTING A HISTOGRAM

Histograms are a useful method to identify data correlation and bell curve distributions.
Picture them as a way to set intervals for data, then collect items in those ranges;

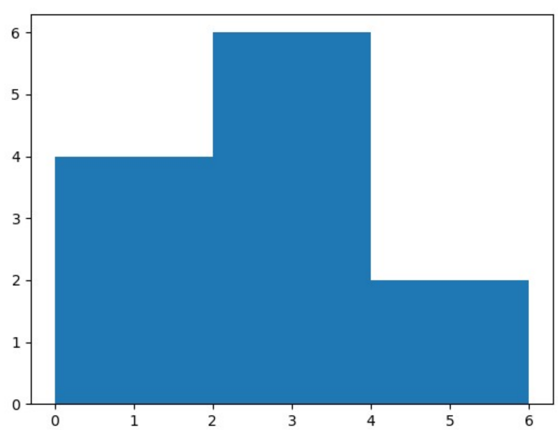


plt.hist(values, bins)

There are many other variables and values a Histogram can contain. Check the help for more.

E.g.

```
import matplotlib.pyplot as plt
list_1 = [0,0.6,1.4,1.6,2.2,2.5,2.6,3.2,3.5,3.9,4.2,6]
plt.hist(list_1, bins=3)
plt.show()
```



PLOTTING A BAR FUNCTION

We will use these functions;

sns.barplot(x='header', y='header', data=totals))
To create a bar plot, putting two values against one another by total

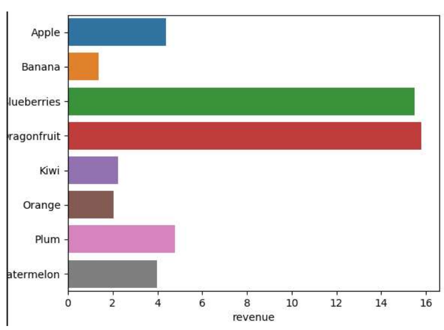
plt.show()
To show the plotted graph

Plt.savefig('file path.png')
To save the plotted graph as an image

```
E.g.
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)
table_1 = workbook_1.parse('Sheet1')
prices_1 = workbook_1.parse('Price')
colors_1 = workbook_1.parse('Color')
combined_1 = colors_1.merge(prices_1, on='name', how='left')
combined_2 = table_1.merge(combined_1, left_on = 'product_name', right_on = 'name', how = 'left')
combined_2['revenue']=combined_2['quantity_purchased']*combined_2['price_usd']
groups_1 = ['store', 'quantity_purchased','name', 'color','price_usd']
combined_3 = combined_2.drop(groups_1, axis=1)
combined_3 = combined_3.groupby('product_name', as_index=False).sum()
bplot_1 = sns.barplot(x = 'revenue', y = 'product_name', data=combined_3)
plt.savefig('Image.png')
```

```
>>> %Run Pandas.py

  product_name  revenue
0        Apple     4.40
1       Banana     1.38
2  Blueberries    15.48
3  Dragonfruit    15.81
4         Kiwi     2.24
5       Orange     2.04
6         Plum     4.80
7   Watermelon     3.98
```



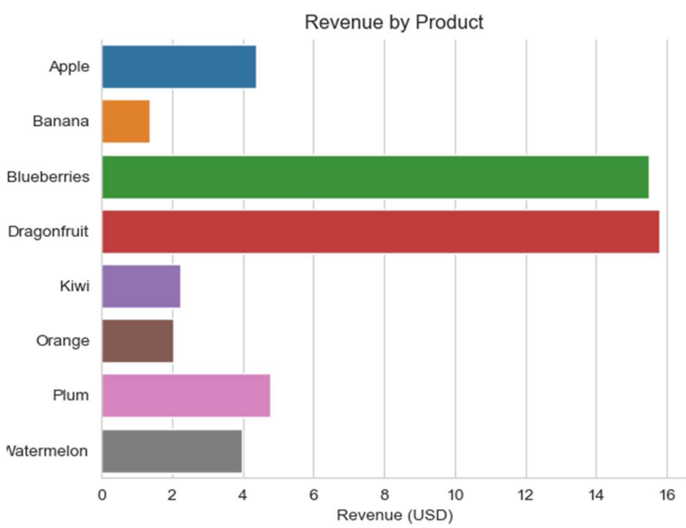
SEABORN STYLES

By default, plots aren't very attractive to look at, we need to format them first.

sns.despine()
Remove the top and right axis' of a bar plot

sns.set_style('whitegrid')
options: dict, None, or one of {darkgrid, whitegrid, dark, white, ticks}
Set a style for the plot before creation

```
E.g.
…
sns.set_style("whitegrid")
sns.barplot(x = 'revenue', y = 'product_name', data=combined_3)
plt.title('Revenue by Product')
plt.xlabel('Revenue (USD)')
plt.ylabel('Product')
sns.despine()
plt.savefig('Image.png')
```



HUE TO RESULTS

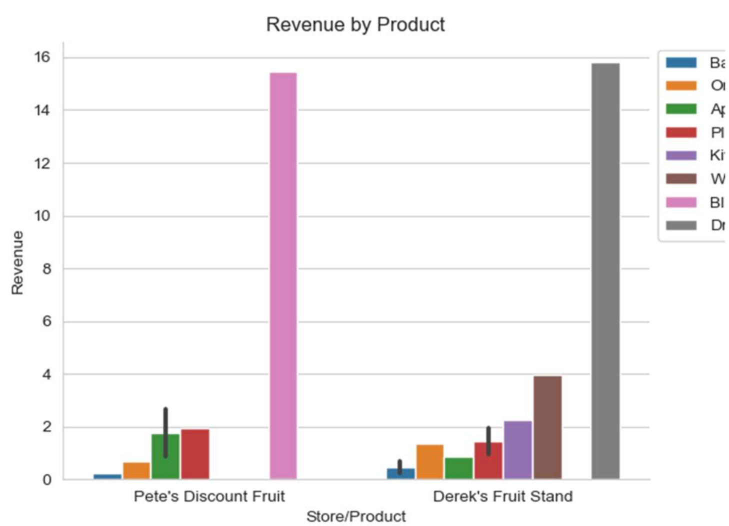
Results on an axis can be further divided using the Hue layer.

```
sns.barplot(x='header', y='header', data=totals, hue='header')
```

The bounding box can be relocated outside the graph using a tuple;
plt.legend(bbox_to_anchor=(1,1))

E.g.

```
print(combined_3)
sns.set_style("whitegrid")
sns.barplot(x = 'store',
        y = 'revenue',
        data=combined_3,
        hue = 'product_name')
plt.title('Revenue by Product')
plt.xlabel('Store/Product')
plt.ylabel('Revenue')
plt.legend(bbox_to_anchor=(1,1))
sns.despine()
plt.savefig('Image.png')
```



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

doc_1 = r'D:\02 Work\02.0D Crone\Python\Pandas\Sample for Python.xlsx'
workbook_1 = pd.ExcelFile(doc_1)

table_1 = workbook_1.parse('Sheet1')
prices_1 = workbook_1.parse('Price')
colors_1 = workbook_1.parse('Color')

combined_1 = colors_1.merge(prices_1, on='name', how='left')
combined_2 = table_1.merge(combined_1, left_on = 'product_name', right_on = 'name', how = 'left')
combined_2['revenue']=combined_2['quantity_purchased']*combined_2['price_usd']

groups_1 = ['name', 'color','price_usd', 'quantity_purchased']
combined_3 = combined_2.drop(groups_1, axis=1)

sns.set_style("whitegrid")
sns.barplot(x = 'store',
        y = 'revenue',
        data=combined_3,
        hue = 'product_name')

plt.legend(bbox_to_anchor=(1,1))

plt.title('Revenue by Product')
plt.xlabel('Store/Product')
plt.ylabel('Revenue')
sns.despine()

plt.savefig('Image.png')
```

PILLOW (PYTHON IMAGE LIBRARY)

VERSION 0.1

(Last edited 9/9/19)

(https://pillow.readthedocs.io/en/4.0.x/handbook/tutorial.html)

PACKAGE – PILLOW

Usually we import the ‘Image’ package from Pillow itself rather than Pillow.

from PIL import Image

Pillow is a package focused on processing, reading, manipulating and generating images.

SET AN IMAGE TO AN OBJECT

Images are assigned to variables the same way as other data types.

Variable = Image.open(‘image file’)

E.g.

```
from PIL import Image
im = Image.open('dncthhpsgky21.png')
```

IMAGE SIZE AND RESIZING

Image size can be read as a tuple, or in separate dimensions using the functions:

Variable.width	the image width in pixels
Variable.height	the image height in pixels
Variable.size	the width and height returned as a tuple (w,h)

E.g.

```
from PIL import Image
im = Image.open('dncthhpsgky21.png')
print(im.size)
print(im.width)
print(im.height)
```

```
>>> %Run 'rgb 2.py'
  (256, 256)
  256
  256
```

Image objects can be resized using the function:

Variable.resize((w, h), Image.ANTIALIAS)

E.g.

```
from PIL import Image
im = Image.open('dncthhpsgky21.png')
im = im.resize((300, 600), Image.ANTIALIAS)
print(im.size)
print(im.width)
print(im.height)
```

```
>>> %Run 'rgb 2.py'
  (300, 600)
  300
  600
```

IMAGE MODES AND CONVERSION

We can check an images file type extension using:

Variable.format

We can also check an images color mode using

Variable.mode

A list of colour modes:

1	(1-bit pixels, black and white, stored with one pixel per byte)
L	(8-bit pixels, black and white)
P	(8-bit pixels, mapped to any other mode using a colour palette)
RGB	(3x8-bit pixels, true colour)
RGBA	(4x8-bit pixels, true colour with transparency mask)
CMYK	(4x8-bit pixels, colour separation)
YCbCr	(3x8-bit pixels, colour video format)
I	(32-bit signed integer pixels)
F	(32-bit floating point pixels)

We can convert our image to another mode using:

Variable.convert(‘mode)

E.g.

```
from PIL import Image
im = Image.open('dncthhpsgky21.png')
print(im.mode)
print(im.format)
im = im.convert('RGB')
print()
print(im.mode)
```

```
>>> %Run 'rgb 2.py'

  P
  PNG

  RGB
```

LOADING IMAGE DATA

In order to read image files in detail, we typically use the load function.

Variable.load()

If we process an image, the image will also be loaded by default, so we do not need to use this if we process an image using Pillow before reading the image with a non-pillow function or method.

READING PIXEL VALUES

We can read pixel values using:

Variable.pixels((x, y))

Note that the pixel value is given as a tuple, hence double bracketed parameters.
If we want to get these pixels as a list, we need to use iteration at an X level upon Y.

E.g.

```
from PIL import Image
im = Image.open('dncthhpsgky21.png')
im = im.convert('RGB')

im_width = im.width
im_height = im.height

print('IMAGE SIZE: '+str(im_width)+'x'+str(im_height))
print('PIXELS CALC = '+str(im_width*im_height))

pixel_list = []

for pixel_y in range(im_height):
    for pixel_x in range(im_width):
        pixel_list.append(im.getpixel((pixel_x, pixel_y)))

print('LIST TYPE: '+str(type(pixel_list)))
pixel_count = len(pixel_list)
print('PIXELS ITERATED = '+str(pixel_count))
print('LIST STRUCTURE:')
print(pixel_list[0:9])
```

```
>>> %Run 'rgb 2.py'

  IMAGE SIZE: 256x256
  PIXELS CALC = 65536
  LIST TYPE: <class 'list'>
  PIXELS ITERATED = 65536
  LIST STRUCTURE:
  [(191, 222, 246), (191, 222, 246), (191, 222, 246), (191, 222, 246), (191, 222, 246), (191, 222,
  246), (191, 222, 246), (191, 222, 246), (191, 222, 246)]
```