Introducing Python with Jupyter

- the idea behind notebooks is that DOCUMENTATION comes first
- most of the area is dedicated to notes, and graphs, plots, etc.
- *few* lines of code , overall

- cf. a std. software program, ~10k+
- data science nb, ~50 to 100 lines
- if you need lots of lines of code... then put those in an external file
- and the notebook is just the output

IMPORTANT:

- run order matters
- must re-run earlier cells if later ones depend on them
- jupyter lists run order next to cell

keyboard shortcuts:

- ctrl enter -- to run
- ESC -- enter command mode
- enter -- enter edit mode

in command mode:

- arrows to move up down
- a create cell abve
- b create cell below
- m to make cell markdown (text)
- y to make cell code
- dd to delete cell

In [4]:

```
print("Hello World")
```

```
Hello World
```

In [5]:

```
import numpy as np
```

In [7]:

```
np.random.random()
```

Out[7]:

```
0.4102587465055618
```

Python Overview

- data
- variables
- operations
- simple functions
- objects
- defining functions
- data structure operations
- libraries
- conditions
- loops
- functional programming (iteration and lambda)

- (appendix: classes, oop)
- later on: numpy, pandas, matplotlib, sklearn

Data

In [12]:

```
print("HI") # print = output to screen
```

```
HI
```

In [8]:

```
print(5)
print(5.0)
print("5")
print(True)
print(["Lovers", "Haters", "Needers"])
```

```
5
5.0
5
True
['Lovers', 'Haters', 'Needers']
```

In [11]:

```
print(type(5))
print(type(5.0))
print(type("5"))
print(type(True))
print(type(["Lovers", "Haters", "Needers"]))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'list'>
```

Variables

In [29]:

```
name = "Michael"
path = "C:\new\trusted"
rpath = r"C:\new\trusted"  # raw string


age = 29
height = 1.81
hobbies = ["Machine Learning", "Programming"]
is_alive = True

fmesg = f"{name} is {age} and {height * 100} cm" # formatted string

print(name, age, height)  # print with multiple args = spaced output
print(hobbies)
print(is_alive)

print(path)
print(rpath)

print(fmesg)
```

```
Michael 29 1.81
['Machine Learning', 'Programming']
True
C:
ew      rusted
C:\new\trusted
Michael is 29 and 181.0 cm
```

In [14]:

```
age += 1

age   # the last line of a jupyter cell is always printed
```

Out[14]:

```
30
```

Operations

In [33]:

```
print( "Michael " + "Burgess")
print( 2 ** 3 )    # 2 * 2 * 2 = 2^3
print( "-" * 3 )
print( "@" in "michael.burgess@qa.com")
print( True and False )
print( ["Cake", "Cream"] + ["Flour", "Sugar"] )
print( 1.14 <=  2.13 )
```

```
Michael Burgess
8
---
True
False
['Cake', 'Cream', 'Flour', 'Sugar']
True
```

Simple Functions

output = fn(input)

returnValue = procedure(requirements)

makes a new value

these are algorithms, not "relationships between mathematical variables"

In [16]:

```
name = "Michael"

print(name)          # output the value of name
print( id(name) )    # output the memory location of the value of name
print( type(name) )  # output the type of the value of name
print( len(name) )   # output the length of (the value of) name
```

```
Michael
2575994870112
<class 'str'>
7
```

Objects

In [17]:

```
# data.operation(requirements)
# obj.method(parameters)
# ask name to upper() itself
# ask name if it startswith(M)

print( name.upper()  )
print( name.lower()   )
print( name.startswith("M") )
print( name.endswith("M") )

# all data in python is an object
# objects are data structures:  values (properties), types (class), id, methods
```

```
MICHAEL
michael
True
False
```

In [51]:

```
dir(name)[-5:]
```

Out[51]:

```
['swapcase', 'title', 'translate', 'upper', 'zfill']
```

Exercise 1:

- define variables of each type mentioned
- they should describe you (name, age, location, etc.)
- print these out
- print all strings in upper case
- print whether your age is over 18
- print 10 dashes

Defining Functions

- algorithm can be used to calculate the value of a mathematical function...
- $error(y_p, y_o) = (y_p - y_o)^2$ (known as the MSE, or, Mean Square Error)
- `def error(pred, obv)` LHS of math
- `return (pred - obv)^2` RHS of math (return aprox., =)
- return actually means store calcuated value in memory

In [24]:

```
def error(pred, obv):
    return (pred - obv) ** 2

error(3, 3.3)
```

Out[24]:

```
0.0899999999999999
```

- indendation groups operations together
- def defines a function
- parameters are listed after the function name
- one new line after the definintion ends the def.
- notice colon before indentation

In [5]:

```
# functions = procedures
# can also not return anything

def show_results(results):
    print("-" * 10)
    print(results)
    print("-" * 10)

show_results([12, 12, 15])   # writes to screen, but has no return value

def distance(x1, x2):
    return (x2 - x1) ** 2   # euclidean distance, aka. L2 norm


dist = distance(10, 12)
print(dist * 1.1)  # calculated value can be stored in variable


rtn = show_results([10, 12])

print(type(rtn))
print(rtn) # nothing is stored here, no return value
```

```
----------
[12, 12, 15]
----------
4.4
----------
[10, 12]
----------
<class 'NoneType'>
None
```

Exercise 2:

define a function called :

- mean which takes three parameters and returns their mean
- cube which cubes its first argument
- is_adult which says whetehr its first argument is more than 18

- define three variables:
- mean_ages which is mean of 18,18,20
- two_later which is 2 cubed
- teen_is_adult which is whether an age of 15 is adult

- define function show()
- which prints the three variables above

In [7]:

```
def mean(x, y, z):
    return (x + y + z)/3

def cube(x):
    return x ** 3

def is_adult(age):
    return age >= 18


def show(m, c, a):
    print("mean:", m)
    print("cube:", c)
    print("age:", a)

mean_ages = mean(18,18,20)
two_late = cube(2)
teen = is_adult(15)

show(mean_ages, two_late, teen)
```

```
mean: 18.666666666666668
cube: 8
age: False
```

Data Structures

- strings - groups of characters
- lists - ordered groups of data where each element is indexed by an int
- sets - unordered groups of data where there is no indexing
- tuples - uneditable (immutable) groups of data where elements are int-indexed
- dictionaries - groups of data where indexes are chosen by you

In [1]:

```
# strings

quote = "Be the change you wish to see in the world!"

print( quote[0] )   # first
print( quote[1] )   # second
print( quote[-2] )  # second from last
print( quote[-1] )  # last
```

```
B
e
d
!
```

In [29]:

```
print( quote[0:2] )  # zero until postn-2
```

```
Be
```

In [30]:

```
print( quote[0:-6] ) # beginning until -6th postn
```

```
Be the change you wish to see in the
```

In [2]:

```
print( quote[0:-6]  + " bed" )
```

```
Be the change you wish to see in the  bed
```

In [34]:

```
print(quote[:2])    # leave off start postn = zero
print(quote[-6:] )  # leave off end postn = end of string
```

```
Be
world!
```

In [38]:

```
# tuple

point = (10, 20, 30)

print( point[0] )
print( point[1] )
print( point[-1] )

print( point[0:2] ) #slice, as with strings

# point[0] = 15 # error: not allowed to overwrite

# technically, () not required...

address = "OldSt", "London"

print(address)
```

```
10
20
30
(10, 20)
('OldSt', 'London')
```

In [45]:

```
# lists
# y target customer satisfaction
# x customer features
# (days-since-first-purchase, total-spent, nearest-store, addresss)
#

x = [300, 1000, "London", ("Old Street", "London")]

print(x)
print(len(x))

print(x[-1])
print(len(x[-1]))
```

```
[300, 1000, 'London', ('Old Street', 'London')]
4
('Old Street', 'London')
2
```

In [46]:

```
x.append(1)
x
```

Out[46]:

```
[300, 1000, 'London', ('Old Street', 'London'), 1]
```

In [47]:

```
x.pop()
```

Out[47]:

```
1
```

In [48]:

```
print(x)
x.insert(0, 1) # insert at postn 0, the element 1

print(x)
```

```
[300, 1000, 'London', ('Old Street', 'London')]
[1, 300, 1000, 'London', ('Old Street', 'London')]
```

Using Lists in Functions

In [20]:

```
def error(y_pred, y, i):
    return (y_pred - y[i]) ** 2
```

In [22]:

```
y = [2, 3, 5, 8]
guess = 2.2

error(first_guess, y, 1)   #   (2.2 - 3) ** 2
```

Out[22]:

```
0.6399999999999997
```

Exericse 3: Lists

- define a list "cart" which is a shopping cart
- add several items to it
- print out the first, last and middle two items
- insert a new item at the start
- print the whole list

Dictionaries

- key-value data structures
- where the keys are defined by you (generally strings)

In [ ]:

```
user = {
    "name": "michael",
    "age": 29,
    "location": "uk"
}

print(user["name"])     # use string keys to look up value rather than int index
print(user["age"])
print(user["location"])
```

In [8]:

```
# data science example: labelling for Fraud|NotFraud
# dict keys can be lots of diff. thigns, not just strings...
# but must be unique!


# key = (age, days-since-purchase-of-insurance)
user = {
    (18, 13) : "Fraud",
    (60, 300) : "NotFraud"
}

user[(18, 13)]
```

Out[8]:

```
'Fraud'
```

In [13]:

```
# dictionaries more commonly are more like matrices...

users = {
    "age-at-purchase": [18, 60],
    "days-from-purcahse": [13, 300]
}

ages = users['age-at-purchase']

sum(ages)/len(ages)
```

Out[13]:

```
39.0
```

In [17]:

```
import pandas as pd
table = pd.DataFrame(users)
# dictionaries can be easily converted to "DataFrames" (table-like data structur
e)

table
```

Out[17]:

	age-at-purchase	days-from-purcahse
0	18	13
1	60	300

In [18]:

```
table['age-at-purchase']
```

Out[18]:

```
0    18
1    60
Name: age-at-purchase, dtype: int64
```

Libraries

- import to include a library
- == python file which defines some functions (etc.)

In [19]:

```
import os
os.listdir('.')
```

Out[19]:

```
['.ipynb_checkpoints',
 'Calculus.md',
 'data',
 'ETL.ipynb',
 'LinearAlgebra.md',
 'ML-CaseStudy-LogReg.ipynb',
 'Python-Introduction.ipynb',
 'PythonOverview.ipynb',
 'Statistics.md']
```

In [20]:

```
import sys
sys.platform
```

Out[20]:

```
'win32'
```

In [30]:

```
import re
print(quote)
re.findall(r"\w+", quote)  # r will escape all backslashes, so they arent interp
red as, eg., new lines
```

```
Be the change you wish to see in the world!
```

Out[30]:

```
['Be', 'the', 'change', 'you', 'wish', 'to', 'see', 'in', 'the', 'wo
rld']
```

In [39]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# std. aliases which make using libs easier

fast_array = np.array([1, 2, 3])
table = pd.DataFrame({"ages": [10, 18, 30], "weights": [50, 70, 80]})
```

In [32]:

```
fast_array
```

Out[32]:

```
array([1, 2, 3])
```

In [33]:

```
table
```

Out[33]:

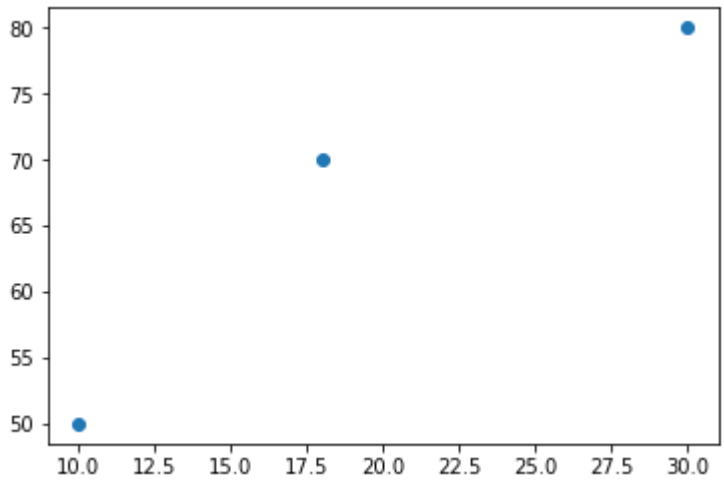
	ages	weights
0	10	50
1	30	80

In [36]:

```
plt.scatter(table["ages"], table["weights"])
```

Out[36]:

```
<matplotlib.collections.PathCollection at 0x1ff9c43dba8>
```



Dropping Prefixes

In [43]:

```
from os import listdir
from numpy.random import random as rn  # mathematicians like short names, makes
 math clearer

listdir('.')  # this is os.listdir
```

Out[43]:

```
['.ipynb_checkpoints',
 'Calculus.md',
 'data',
 'ETL.ipynb',
 'LinearAlgebra.md',
 'ML-CaseStudy-LogReg.ipynb',
 'Python-Introduction.ipynb',
 'PythonOverview.ipynb',
 'Statistics.md']
```

In [42]:

```
rn() # numpy.random.random
```

Out[42]:

```
0.40920597473805
```

Exercise 4: Libraries

- import pandas, matplotlibs's pyplot with the standard alises
- define a dictionary called "facebook_users"
- keys are columns: "uid, followers, friends, total_comments, total_likes"
- values are lists (rows): eg., a list of user ids

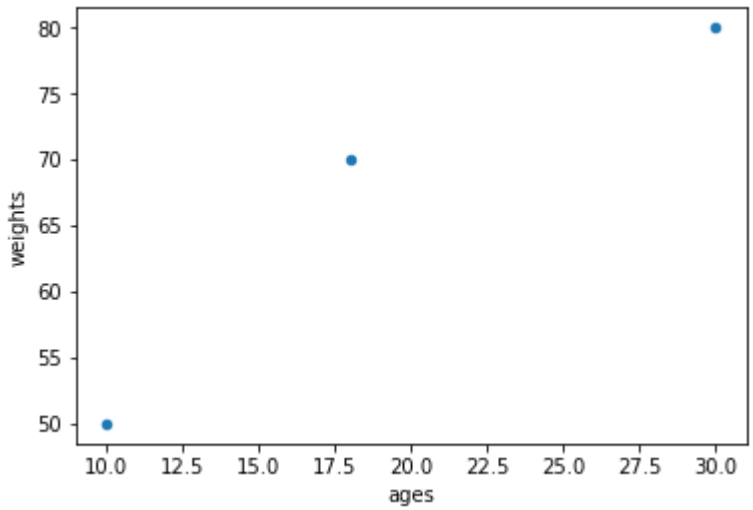
- create a pandas data frame from this dictionary
- use matplotlib to draw a scatter of the dictionary: followers vs total_likes

In [45]:

```
table.plot.scatter("ages", "weights")
```

Out[45]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1ff9db51b38>
```



Control Flow

In [47]:

```
user_age = 18

if user_age > 65:                          # colons
    print("See Retirement Plans")          # indentation
elif user_age > 21:                        # keyword, elif
    print("See Vocation Plans")
elif user_age > 13:
    print("See Education Plans")
else:
    print("See your mother!")
```

```
See Education Plans
```

In [50]:

```
# while loops are rare, usually bad -- repeating

ratings = [5,5,6,7,8,1]

while len(ratings) > 0:
    print(ratings.pop())    # remove last one
```

```
1
8
7
6
5
5
```

In [49]:

```
ratings
```

Out[49]:

```
[]
```

In [52]:

```
# for loop -- data processing loop

ratings = [5,5,6,7,8,1]

for element in ratings:      # for name-of-each-element  in source-data-input
    print(element)           # algorithm for processing each-element

ratings
```

```
5
5
6
7
8
1
```

Out[52]:

```
[5, 5, 6, 7, 8, 1]
```

Type Conversions

In [54]:

```
# iterators -- like data structures, but whole data not stored...

ten = range(0, 10)

print(ten)
```

```
range(0, 10)
```

In [57]:

```
for i in ten:       # the range gives a number each go around
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [56]:

```
numbers = list(ten) # collects all data from ten into list, which stores all in
 memory

numbers
```

Out[56]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [58]:

```
age = "18"

age < 20
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-58-387956351189> in <module>
      1 age = "18"
      2
----> 3 age < 20

TypeError: '<' not supported between instances of 'str' and 'int'
```

In [59]:

```
int(age) < 20
```

Out[59]:

```
True
```

In [60]:

```
str(5) * 2  # "5" * 2
```

Out[60]:

```
'55'
```

In [61]:

```
dict( [ ("name", "Michael"), ("age", 29 )])
```

Out[61]:

```
{'name': 'Michael', 'age': 29}
```

Functional Programming

- passing around functions
- internal iteration

In [68]:

```
def transform(fn, data):
    out = []
    for e in data:
        out.append(fn(e))

    return out


films = ["Annie Hall", "American Beauty", "Manhatten"]
transform(len, films)
```

Out[68]:

```
[10, 15, 9]
```

In [69]:

```
# map(fn, data)  -- return value is an iterator, so gives one piece at a time

print(len(films)) # 3

list(   map(len, films)    )
```

```
3
```

Out[69]:

```
[10, 15, 9]
```

In [72]:

```
# what about len(film) 10 ?

# map( len... > )  nope


list(

    map(  lambda f : len(f) > 10,     # lamda input : return-value
        films)

)

# use lambda to create functions when the don't exit
# and you dont want to write a def.
```

Out[72]:

```
[False, True, False]
```

In [75]:

```
# LIST COMPREHENSION:


# better to avoid lambda if possible..

              # expr-for-new-el  where element comes-from   data-source
new_list = [ len(film) > 10      for    film     in            films ]
new_list
```

Out[75]:

```
[False, True, False]
```

Exercise 6: Functional Programming

- define a list of names
- use a list comprehension to make a new list of
- uppercase names
- lowercase names
- booleans, where True if the length of the name is more than 5 characters

Appendix: Classes

In [63]:

```
class Customer:
    GOOD_RATING = 7                          # class scope ~ static

    def __init__(self, name, rating):        # constructor
        self.name = name
        self.rating = rating

    def __str__(self):                      # self == this
        return name.upper()

    def is_good(self):
        return self.rating > Customer.GOOD_RATING


michael = Customer("Michael", 8)  # no new keyword

print(michael)                  # converts to string using str(), which calls ._
_str__()
print(michael.is_good())
```

```
MICHAEL
True
```