01_PythonIntroduction

May 17, 2022

0.1 Data, Analytics & AI

1 Python Programming

1.1 Part 1: Why Python?

(because it has lots of useful libraries…)

The `import` keywords loads external "libraries" (systems of tools) that can solve various problems…

very fast numerical programming

[6]:
```
import numpy
```

very fast spreadsheet/tabular data processing

[7]:
```
import pandas
```

very fast visualization…

[8]:
```
import seaborn
```

very fast predictive analysis ("machine learning")

[9]:
```
import sklearn
```

…some problem-specifc ones…

quite good network analysis

[10]:
```
import networkx
```

Eg., loading the natural language toolkit…

[11]:
```
import nltk
```

1.2 Quick Library Demos

1.2.1 NumPy for fast numerical computing

10 random numbers with a mean of c. 5, and a variation +- c. 2...

[12]:
```
numpy.random.normal(5, 2, 10) # library = numpy
```

[12]:
```
array([7.3394182 , 4.45010227, 2.04679137, 2.22313196, 4.88148774,
       9.18114105, 6.94856322, 4.77116333, 6.07060989, 2.63717942])
```

1.2.2 Pandas for SQL-like, Spreadsheet-Like Operations

[15]:
```
df = pandas.read_excel('LIVE_ExcelDemoFile.xlsx'); df # library = pandas
```

[15]:

	Age	Years	Profit
0	18	0	100
1	20	1	200
2	31	10	1000
3	80	50	10000

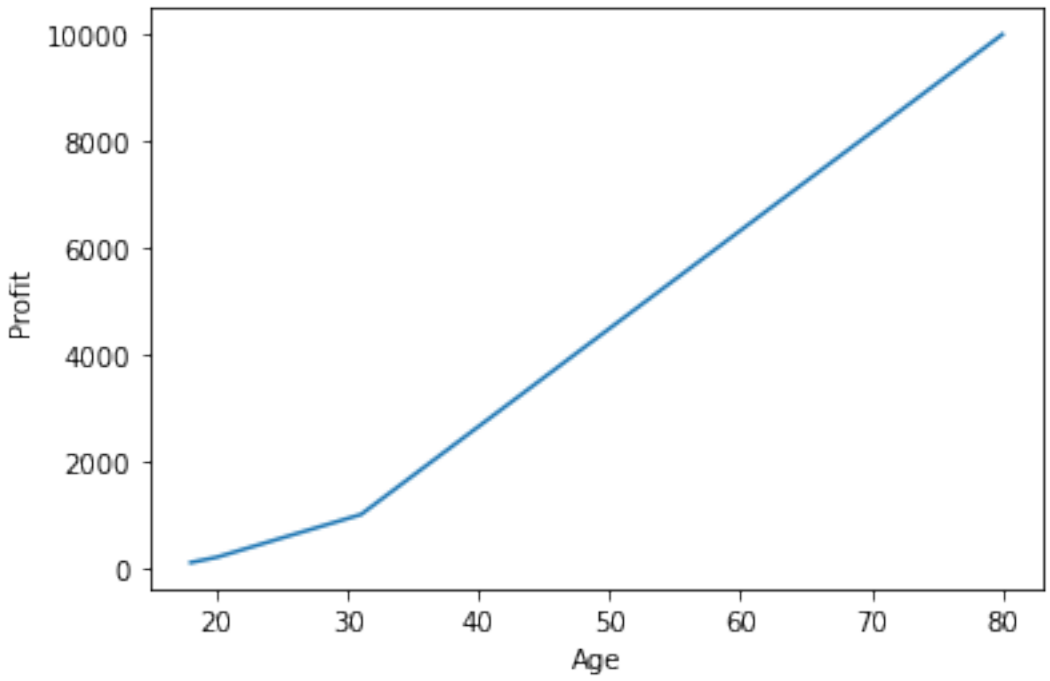
[16]:
```
df['Profit'].mean()
```

[16]:
```
2825.0
```

1.2.3 Seaborn for Visuals

[17]:
```
seaborn.lineplot(x=df['Age'], y=df['Profit']) # library = seaborn
```

[17]:
```
<AxesSubplot:xlabel='Age', ylabel='Profit'>
```



1.2.4 Sklearn for Predictive Analytics

[18]:
```
from sklearn.linear_model import LinearRegression
```

find the relationship...

[19]:
```
model = LinearRegression().fit(df[['Age']], df['Profit']) # sklearn
```

use it to predict some profits...

[20]:
```
model.predict([
    [40],
    [60]
])
```

```
C:\Users\Thomas Holmes\.conda\envs\ppds\lib\site-packages\sklearn\base.py:450:
UserWarning: X does not have valid feature names, but LinearRegression was
fitted with feature names
  warnings.warn(
```

[20]:
```
array([3276.19094585, 6557.57964296])
```

Aside: the underlying formula...

[21]:
```
model.coef_
```

[21]:
```
array([164.06943486])
```

[22]:
```
model.intercept_
```

[22]:
```
-3286.586448367687
```

Sklearn has determine the relationship between x and y is...

$y_{profit} = 164x_{age} - 3287$

1.3 What is python (vs. its libraries)?

The receipe for using a library is...

```
import library
saved_output = library.tool(input_for_tool)
```

Python *itself* is a language of built-in keywords and operations...

Python is really slow... when we're using python keywords ("built in operations") they run slower than the fast libraries above...

[23]:
```
age = 50 # ASSIGN

if age > 35: # IF, COMPARE
    print("LEAVE") # THEN, PRINT
else:
    print("STAY") # ELSE, PRINT


data = [1, 2, 3, 4] # ASSIGN LIST OF VALUES

for d in data: # PROCESS EACH VALUE IN LIST
    print(d)   # PRINT EACH VALUE
```

```
LEAVE
1
2
3
4
```

However when automating systems, our approach will be to *glue* the libraries above "together" using these python keywords...

[24]:
```
answer = numpy.random.normal(10, 10, 100).mean() # simulate some profits
excel = pandas.read_excel('LIVE_ExcelDemoFile.xlsx') # obtain some data

if excel['Profit'].mean() > answer:  # compare simulation with data
    print("GOOD YEAR")
else:
```

```
    print("BAD YEAR")
```

```
GOOD YEAR
```

The majority of python programming is just gluing libraries together… most of the problem is being solved by the libraries… our "glue" can be very slow and still not matter..

1.4 Part 2: Python Programming

- assignment
- data types
  - int
  - float
  - lists
  - dicts
    - relationships
  - str
    - text
- text formatting
- decisions
  - if, else,
- processing data
  - for

1.5 "Data Points"

[25]:
```
age = 31
```

[26]:
```
age
```

[26]:
```
31
```

[27]:
```
name = "Michael" # quotes = text
```

[28]:
```
name
```

[28]:
```
'Michael'
```

[29]:
```
weight = 89.3
```

[30]:
```
weight / 2
```

[30]:
```
44.65
```

On machines we have to distinguish between imprice partial numbers, floats, and precise whole numbers call ints…

[31]:
```
type(weight)
```

[31]:
```
float
```

[32]:
```
type(age)
```

[32]:
```
int
```

[33]:
```
0.1 ** 3
```

[33]:
```
0.0010000000000000002
```

[34]:
```
0.1 * 0.1 * 0.1
```

[34]:
```
0.0010000000000000002
```

..aside: sometimes this impresicion can be useful. Eg., in certain types of predictive analytics (on images, etc.) predictions can be faster if they are lower precision.

1.6 "Data Sets"

Lists are like single column: entires, and an index

[35]:
```
ages = [18, 30, 45, 70] # square brackets and comma
```

[36]:
```
type(ages)
```

[36]:
```
list
```

Find the element at *position* 0 in ages,

[37]:
```
ages[0] # accessing elements in a list uses a[]
```

[37]:
```
18
```

[38]:
```
ages[0] # indexes start at zero
```

[38]:
```
18
```

Dictionaries are *relationships* between values...

[39]:
```
loves = { # braces

    # KEY     :  VALUE
    'Michael' : 'Pork'   , # comma
    'Alice'   : 'Cheese' ,
    'Eve'     : 'Wine'

}
```

Accessing elements (nearly) always uses square brackets...

[40]:
```
loves['Michael'] # use the tag or "KEY" to find the VALUE
```

[40]:
```
'Pork'
```

If we combine dictionaries and lists together, we get tables...

[41]:
```
table = {
    'Age'   : [18, 19, 20, 31],
    'Profit': [100, 200, 350, 1000]
}
```

When we tag lists with KEYs in a dictionary, this creates something like a table..

The column is named...

[42]:
```
table['Age']
```

[42]:
```
[18, 19, 20, 31]
```

And we can get the first entry just using `0` as above...

[43]:
```
table['Age'][0]
```

[43]:
```
18
```

1.7 Basic Operations

- Indexing data
  - finding elements within data using an index

[44]:
```
#       -4  -3   -2  -1
#       0   1    2    3
ages = [18, 30, 45, 70]
```

[45]:
```
ages[0]
```

[45]:
```
18
```

[46]:
```
ages[-1]
```

[46]:
```
70
```

[47]:
```
ages[-2]
```

[47]:
```
45
```

[48]:
```
person = {
    'name': 'Michael',
    'age': 31
}
```

with dictionaries, indexes are (typically) text, and to find an elemennt in a dictionary we use this text KEY…

[49]:
```
person['name']
```

[49]:
```
'Michael'
```

- printing data

by default, jupyter will print the last entry in a cell…

[50]:
```
1
2
5
```

[50]:
```
5
```

[51]:
```
print("Michael")
print("Michael")
print("Michael")
print("Michael")
```

```
Michael
Michael
Michael
Michael
```

The print command takes as many inputs as you like, separated by a comma, and each input is printed surrounded by a space…

[52]:
```
print(person['name'], 'is', person['age'], 'years old')
```

```
Michael is 31 years old
```

[53]:
```
print("Michael's Bio")
print() # empty = blank line
print(name, 'is', age, 'and weights', weight, 'kg')

print(name, 'loves', loves[name]) # name = 'Michael'
print(name, 'has friends who are', ages) # all of the list
print(name, 'has an old friend who is', ages[-1] ) # last part
print(name, 'likes people in the range', ages[-1] - ages[0], 'years')
```

```
Michael's Bio
```

```
Michael is 31 and weights 89.3 kg
Michael loves Pork
Michael has friends who are [18, 30, 45, 70]
Michael has an old friend who is 70
Michael likes people in the range 52 years
```

[54]:
```
difference = ages[-1] - ages[0]
print('michael likes', difference)
```

```
michael likes 52
```

1.8 Exercise (c. 15min)

Consider the company you work for, or otherwise, consider one of the following problem area; and write up a python-based dataset.

E.g. * finance * a dataset for profit/loss on some stocks * retail * a dataset for toys/items in a store * health * a dataset for hr/bp/sleep quality, etc.

- define several lists which represent some possible columns
- define a dictionary which tags some values
- finally, print out elements of your dataset in a report

1.8.1 Hints & Guidance

- exercises are NOT exams
  - the point isn't to answer questions
  - the time is *yours* to spend independently, exploring the topic
    - eg., you may browse online documentation and other resources if you are sure of the content
- (new learners should) start by reading through relevant notebooks
  - run each cell
  - modify according your curiosity
  - see how your changes impact code
  - convince yourself you understand the code
- create a new notebook
  - copy/paste examples across
  - modify until they solve the problem above
- experienced learners should…
  - create a blank notebook and solve the exercise from scratch

2 Python Decision Making

Let's ask the user for their age and city, a float and a string. Then make a holiday decision based on these…

[55]:
```
age = float(input("Age? "))
```

```
Age?  20
```

[56]:
```
age
```

[56]: 20.0

[57]:
```
city = input("City? ")
```

```
City?  Ldn
```

[58]:
```
city
```

[58]: 'Ldn'

Below we use some keywords (`if`, `elif`, `else`; `and`, `in`)...

[59]:
```
if (age >= 18) and (city in ["Leeds", "London"]): # the indent region below is␣
↪a *single* block
    print("let's go in holiday")
    print("let's go in holiday")
    print("let's go in holiday")
    print("let's go in holiday")
elif (age >= 18) and (city == "Paris"):
    print("that's on the red list!")
elif (age <= 18):
    print("stay at home!")
    print("stay at home!")
    print("stay at home!")
else:
    print("contact the foreign office!")
```

```
contact the foreign office!
```

The decision making keywords each consider a condition, and if that condition is `True` then the code below that condition is exceuted.

Only one block of code is executed; the `elif` keyword introduces addition tests, and `else` executes if all tests fail (ie., `False`).

2.1 Comparions

[60]:
```
age >= 18
```

[60]: True

[61]:
```
city in ["Leeds", "London"]
```

[61]: False

[62]:
```
city in ["Paris", "Rome", "Berlin"]
```

[62]: False

```
[63]: city == "Paris"
```

[63]: False

```
[64]: city != "London"
```

[64]: True

To make a case-insensitive comparison, we can convert the value to uppercase before we compare..

```
[65]: city.upper() == "PARIS"
```

[65]: False

2.2 Other Numerical Comparisons

```
[66]: 10 > 50
```

[66]: False

```
[67]: 10 > 5
```

[67]: True

```
[68]: 5 < 10
```

[68]: True

```
[69]: 10 <= 10
```

[69]: True

```
[70]: 10 < 10
```

[70]: False

```
[71]: 10 == 10
```

[71]: True

```
[72]: 10 != 10
```

[72]: False

```
[73]: 10 != 5
```

[73]: True

```
[74]: 5 < 10 < 20 # both 5 < 10 AND 10 < 20.... 10 is between 5, 20
```

[74]: True

2.2.1 Aside:

```
[75]: (5 < 10) and (10 < 20)
```

[75]: True

2.3 Other Text Comparisons

```
[76]: city.startswith("P")
```

[76]: False

```
[77]: city.endswith("don")
```

[77]: False

```
[78]: city.isupper()
```

[78]: False

```
[79]: city.isalpha() # isalphabetical
```

[79]: True

2.4 Other Comparison Operations

```
[80]: "P" in city
```

[80]: False

```
[81]: "L" not in city
```

[81]: False

2.5 Combining Comparions

```
[82]: age
```

[82]: 20.0

[83]:
```
("P" in city) and (age >= 18) # both = conjuction
```

[83]: False

[84]:
```
("P" in city) or (age >= 18) # either = disjunction
```

[84]: True

[85]:
```
not (age >= 18) # not = negation
```

[85]: False

[86]:
```
(age < 18) # the negation of a numerical comparison, can often be written␣
↪without `not`
```

[86]: False

[ ]:
```

```

2.6 Decision-Making with Python Keywords

[87]:
```
y_rating = None # None is a special value which typically means "missing" /␣
↪"nothing"
x_age = 18

if x_age >= 18:
    print("we would like you to rate the film!")
```

```
we would like you to rate the film!
```

[88]:
```
if 18 <= x_age <= 35:
    y_rating = 7
else:
    y_rating = 3
```

[89]:
```
y_rating
```

[89]: 7

[90]:
```
x_age = 40

if 18 <= x_age <= 35:
    y_rating = 7
elif 36 <= x_age <= 65:
    y_rating = 6
else:
    y_rating = 3
```

[91]:
```
y_rating
```

[91]:
```
6
```

2.7 Aside: Conditional Expressions

An expression is a *calucation* which is identical to the value it computes. `if` can be used to *select* a value...

[92]:
```
age
```

[92]:
```
20.0
```

[93]:
```
location = "London" if age >= 18 else "Leeds"
```

[94]:
```
location
```

[94]:
```
'London'
```

[95]:
```
("London" if age >= 18 else "Leeds").upper()
```

[95]:
```
'LONDON'
```

```
choose "London" (if age >= 18) or "Leeds" (otherwise)
```

[96]:
```
( 5 + 3 ) * 2
```

[96]:
```
16
```

[97]:
```
( 2 ** 10 ) if age >= 18 else (2 ** 20 )
```

[97]:
```
1024
```

2.8 Exercise (15 min)

Choose a problem domain. Eg., healthcare: advise on a diet based on user survey responses; questions, eg., how many sweets do you eat a day? etc.

2.8.1 Part 1

Conduct a survey (of your design) using the `input()` function. Produce a formatted report of your user's answers.

The survey should ask questions with floating-point answers and text answers.

HINT: `float(input("Question? "))`

For formatting, have a go at using `print(f"text... {variable}")`

2.8.2 Part 2

Considering the survey responses above, *advice* your user on the best options for some problem they are having.

3 Sharing Code with Python

3.0.1 Functions & Libraries

3.0.2 How do I write code which can be shared?

This code is specific (specialized) to a particular dataset,

[98]:
```
x_balance = 1_000
x_age = 32
x_postcode = "SW1 1AA"

y_profit = x_balance * 0.1 + x_age
y_profit
```

[98]: 132.0

We would like to share the formula above and allow anyone to reuse it, regardless of what dataset they are dealing with…

[99]:
```
def profit_formula(x_balance, x_age):
    y_profit = x_balance * 0.1 + x_age

    return y_profit
```

Once this is defined we can *reuse* this formula with a variety of different input arguments,

[100]:
```
profit_formula(1_000, 32) # y_profit = 1_000 * 0.1 + 32
```

[100]: 132.0

[101]:
```
profit_formula(2_000, 65)
```

[101]: 265.0

[102]:
```
profit_formula(3_000, 65)
```

[102]: 365.0

3.0.3 Syntax

```
def NAME_OF_FUCTION(INPUT_ARGUMENTS, ...):

    OUTPUT = CODE
```

```
    return OUPUT
```

[103]:
```
input_1 = 1000
input_2 = 32

output = profit_formula(input_1, input_2)
```

[104]:
```
output
```

[104]: 132.0

A function can be understood as a rewriting sytem (copy / paste)...

[105]:
```
def calc(x, y):
    return 2 * x + y
```

[106]:
```
calc(10, 20)
```

[106]: 40

Step 1: replace calc(10, 20) with 2 * x + y

Step 2: replace x with 10 replace y with 20

Step 3: run 2 * 10 + 20

[107]:
```
calc(10, 20) == 2 * 10 + 20
```

[107]: True

A function is, on one view, just a simple means to *name* a block of reusable code so you don't have to copy/paste the formula (its code) everywhere.

3.0.4 Other advantages of functions

- "black-box abstraction"
  - you dont need to know how a function works, to use it

I can be a user of functions without knowing how they work,

[108]:
```
"Hello".upper()
```

[108]: 'HELLO'

[109]:
```
profit_formula(10, 20)
```

[109]: 21.0

, this is essential! Programming is involves 1000s of functions, at least.

- functions are easy to share, document, reuse...
  - I can just give a person the name of a function and what inputs it needs

[110]:
```
calc(20, 7)
```

[110]:
```
47
```

3.0.5 How do write code to share it?

The first step to sharing code (ie., making it reusable) is turning it into a function.

Consider the code below, it isn't very reusable...

[111]:
```
hr = float(input("What's your HR?"))
bp = float(input("What's your BP?"))
sleep = float(input("How many hours did you sleep?"))

quality = round( (sleep/12 - hr/120 - bp/200)/3 , 2)

print("Your HR is", hr)
print("Your BP is", bp)
print("Your Sleep was", sleep)
print()
print("Your health quality was calc'd to be", quality)
```

```
What's your HR? 120
What's your BP? 120
How many hours did you sleep? 2

Your HR is 120.0
Your BP is 120.0
Your Sleep was 2.0

Your health quality was calc'd to be -0.48
```

[112]:
```
# ASK USER FOR THEIR DATA
hr = float(input("What's your HR?"))
bp = float(input("What's your BP?"))
sleep = float(input("How many hours did you sleep?"))

# COMPUTE A QUALITY
quality = round( (sleep/12 - hr/120 - bp/200)/3 , 2)

# REPORT ON USER & QUALITY
print("Your HR is", hr)
print("Your BP is", bp)
print("Your Sleep was", sleep)
```

```
print()
print("Your health quality was calc'd to be", quality)
```

```
What's your HR? 180
What's your BP? 20
How many hours did you sleep? 2

Your HR is 180.0
Your BP is 20.0
Your Sleep was 2.0


Your health quality was calc'd to be -0.48
```

[113]:
```
def health_ask():
    hr = float(input("What's your HR?"))
    bp = float(input("What's your BP?"))
    sleep = float(input("How many hours did you sleep?"))

def health_quality():
    quality = round( (sleep/12 - hr/120 - bp/200)/3 , 2)

def health_report():
    print("Your HR is", hr)
    print("Your BP is", bp)
    print("Your Sleep was", sleep)
    print()
    print("Your health quality was calc'd to be", quality)
```

Above we have partitioned the code into three functions (abitary, could be more or fewer, eg., one each for each question).

The problem at this point is that the variables defined within the functions are part of their "defintion", they are *local* to those functions. They are not output'd.

(Aproximately,) the only way of getting data into a function is using the arguments, and out of a function, using `return`.

[114]:
```
def health_ask():
    return {
        'HR'    :   float(input("What's your HR?")),
        'BP'    :   float(input("What's your BP?")),
        'Sleep' :   float(input("How many hours did you sleep?")),
    }

def health_quality(sleep, hr, bp):
    return round( (sleep/12 - hr/120 - bp/200)/3 , 2)

def health_report(sleep, hr, bp, quality):
    print("Your HR is", hr)
```

```
    print("Your BP is", bp)
    print("Your Sleep was", sleep)
    print()
    print("Your health quality was calc'd to be", quality)
```

Above we have *defined* three "receipes" (algorithms, formula, functions…); we have yet to **RUN** them.

[115]:
```
data = health_ask() # save the return'd output
```

```
What's your HR? 4
What's your BP? 4
How many hours did you sleep? 4
```

`data` is the *saved* dictionary which is computed by `health_ask()`, which we RUN previously.

[116]:
```
data
```

[116]:
```
{'HR': 4.0, 'BP': 4.0, 'Sleep': 4.0}
```

We now run `health_quaity` using `data` as its input, and *save* the output (`return`'d value) in `result`,

[117]:
```
result = health_quality(data['Sleep'], data['HR'], data['BP'])
```

Finally we use `health_report` to display all these values.

[118]:
```
health_report(data['Sleep'], data['HR'], data['BP'], result)
```

```
Your HR is 4.0
Your BP is 4.0
Your Sleep was 4.0

Your health quality was calc'd to be 0.09
```

In one go, **using pre-defined functions**

[119]:
```
data   = health_ask()
result = health_quality(data['Sleep'], data['HR'], data['BP'])

health_report(data['Sleep'], data['HR'], data['BP'], result)
```

```
What's your HR? 4
What's your BP? 4
How many hours did you sleep? 4

Your HR is 4.0
Your BP is 4.0
Your Sleep was 4.0
```

```
Your health quality was calc'd to be 0.09
```

Compare this with the original, using **no functions**,

[120]:
```
hr = float(input("What's your HR?"))
bp = float(input("What's your BP?"))
sleep = float(input("How many hours did you sleep?"))

quality = round( (sleep/12 - hr/120 - bp/200)/3 , 2)

print("Your HR is", hr)
print("Your BP is", bp)
print("Your Sleep was", sleep)
print()
print("Your health quality was calc'd to be", quality)
```

```
What's your HR? 4
What's your BP? 4
How many hours did you sleep? 4
```

```
Your HR is 4.0
Your BP is 4.0
Your Sleep was 4.0

Your health quality was calc'd to be 0.09
```

3.1 Why use functions for data analysis?

Mostly, the functions will be written by a *senior* team member (or in another team), and the junior/mid-level analyst will be a *client*.

It is their job simply to learn the vocabulary defined by the software staff, and to learn how these "functions" (words, operations) fit together.

This can be much simpler than learning python.

3.2 How do I share functions?

In jupyter, browse to the right folder (with your notebook), NEW **text file**. Rename so it ends with `.py`.

Cut/paste the function into this file, and input **just the name** (ie., not `.py`).

[121]:
```
import shared
```

[122]:
```
shared.calc(10, 20) # from shared.py
```

[122]: 40

[123]:
```
calc(10, 20) # from the notebook
```

[123]: 40

In practice you shouldnt have functions in both the **module** (file) and the notebook. Whilst sketching you can, but then you share the module only.

[124]:
```
shared.X_DEFAULT
```

[124]: 10

3.3 What are modules?

Modules are plain text files that end with `.py` and contain python code. You can `import` these files which *runs* them, and any terms which are defined within, can be used as `module.term`.

Eg., consider a file called `shared.py`,

```
X_DEFAULT = 10
Y_DEAFULT = 20

def calc(x, y):
    return 2 * x + y
```

If I write, `import shared`, then I can also write,

```
print(shared.X_DEFAULT))
print(shared.calc(5, 10))
```

3.4 Review

Suppose I want to share a `crimes` prediction formula which predicts the number of crimes in an area..

What should I call it?

```
def crimes(  ....   ):
```

What does this formula need as input?

```
def crimes(population, police, wealth):
```

What do I need to compute?

```
    wealth_per_person = wealth/population
    police_per_person = police/population
    baseline_crime = 1000/population


    prediction = baseline_crime - wealth_per_person - police_per_person
```

What's the output? Let's `return`,

```
    return prediction
```

So I write,

[128]:
```
def crimes(population, police, wealth):
    wealth_per_person = wealth/population
    police_per_person = police/population
    baseline_crime = 100_000/population

    prediction = baseline_crime - wealth_per_person - police_per_person

    return prediction
```

This *defines* it (putting it in a book recepies,) – to *RUN* the algorithm, I use the name followed by a comma-separated sequences of its inputs…

[126]:
```
crimes(10_000, 25, 35_000)
```

[126]:
```
6.4975
```

Now let me pull this code out and *share* it.

1. Create a blank text file with a useful name
2. Import the file
3. Use the module name (ie., file name) followed by .crimes

[129]:
```
import crimestats
```

[130]:
```
crimestats.crimes(15_000, 200, 40_000)
```

[130]:
```
3.986666666666667
```

3.5 Exercise (30 min)

Let's predict some film ratings; and let's output a report.

[131]:
```
def predict_like_sandler(age, food_cost):
    if age > 18:
        return 'NO'
    elif food_cost > 10:
        return 'YES'
    else:
        return 'NO'
```

[132]:
```
predict_like_sandler(10, 15)
```

[132]:
```
'YES'
```

[133]:
```
def film_report(name, age, food_cost):
    like = predict_like_sandler(age, food_cost)

```

```
    print(name, 'likes this film:', like)
```

[134]:
```
film_report('Michael', 32, 10)
```

```
Michael likes this film: NO
```

3.5.1 Questions

- Modify the definition of `predict_like_sandler` add another argument, `run_time`
  - HINT: add an extra argument at the end of def
    - `def .. (..., extra):`
- Modify the middle condition so that the food costs stays high, but also require the run_time to be low
  - HINT: (food_cost > 10) and (….)
- Modify `film_report` to accept a `run_time`
  - HINT: just like your modification to `predict_like_sandler`
  - HINT: it should just pass it to `predict_like_sandler(..., extra)`
  - HINT: maybe you want to print the run time too…
    - ie., `print()` the run_time in the report

3.5.2 EXTRA

- Extract these functions it their own module & import

[ ]:

[ ]: