Programming with python™ PART II

Matthieu Miossec, PhD
Bioinformatics Core @ Wellcome Centre for Human Genetics
https://www.well.ox.ac.uk/people/matthieu-miossec

20/02/2023
AFTERNOON

Part I challenge: Fizzbuzz, a few solutions

"Write a script that takes the numbers 1 through 100, prints *fizz* (to screen) if said number is a multiple of 3, *buzz* if a multiple of 5 and *fizzbuzz* if a multiple of 5 and 3 (…and prints numbers otherwise)."

Did you manage? Here are two solutions…

Only the first True statement is evaluated

```
for i in range(1,101):
    if i % 15 == 0:
        print("fizzbuzz")
    elif i % 3 == 0:
        print("fizz")
    elif i % 5 == 0:
        print("buzz")
    else:
        print(i)
```

```
for i in range(1,101):
    out = ""
    if i % 3 == 0:
        out="fizz"
    if i % 5 == 0:
        out +="buzz"
    if out:
        print(out)
    else:
        print(i)
```

Three <u>independent</u> IF statements here.

Notice += here. This will add "buzz" to either an empty *string* or a *string* that says "fizz“.

var+= num
is shorthand for:
var = var + num

Remember, a non-empty string is **True**.
Quicker than writing out len(out) !=0

- Previously, we looked at built-in data types *string*, *integer*, *float* and *Boolean*.
  - Python also has what are called **data structures**, more specifically *tuples* and *lists* and *dictionaries* (oh my!).

```
a_list = [1, 2, 3]  # This is a list
a_tuple = (4, 5, 6)  # This is a tuple
a_dict = {‘fname': 'John', 'sname': 'Doe', 'age': 42} # This is a dictionary
```

  - These data structures can store the data types we’ve seen so far.

```
days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun') #tuple of string
```

  - Python allows for distinct data types within the same structure.

```
misc = ["sunny", 1, 36.8, True]  # List with string, int, float and boolean
```

  - They can also contain themselves and each other!

```
litup = [('A','a'),('B','b'),('C','c'),('D','d')] # List of tuples
```

type() works on data structures as well (it won’t tell you what’s inside though!)

- *Tuples* and *lists* look superficially the same (apart from the brackets), but in fact behave quite differently.
  - Unlike lists (but like *strings*), **tuples are immutable**

```
weather = ("sun", "clouds", "rain", "snow")
print(weather[1])  # We can access items of the tuple like clouds.
# We can't replace or remove them.
weather[1] = "mist"  # This will cause an error!
```

  - As with strings, we can get a slice of a tuple…

```
print(weather[1:3]) # Notice the output is a tuple ('clouds', 'rain').
print(weather[-1])  # Notice here the output is string 'snow'.
```

  - …determine length, concatenate or multiply tuples into new tuples.

```
print(len(weather)) # 4
print(weather + ("mist", "thunder")) # New tuple w/ mist and thunder
print(weather[1:3]*3)  # sun and clouds repeated 3 times
```

Data Structures (("tuples"), ("within", ("tuples")))

UNIVERSITY OF OXFORD

- Tuples can contain other data structures.
  - If a data structure exists within another, we can still access its items in the following manner.

```
forecast = (("sun", 32.6), ("clouds", 16.2), ("rain", 14), ("snow",-1))

print(forecast[2][1]) # 14
print(forecast[0][0]) # sun
print(forecast[1:3][1]) # ('rain', 14).
```

  - If the result of the last line feels unexpected, think about what you generated in the first instance.

```
slicecast = forecast[1:3]
print(slicecast)  # ("clouds", 16.2), ("rain", 14)
print(slicecast[1]) # ('rain', 14). Same result in more steps.
```

  - Question: Which tuple leads to a False when cast to bool()?

UNIVERSITY OF OXFORD

- We've seen that both *strings* and *tuples* are immutable. What does a **mutable** data structure look like?
  - We now turn to **lists**.

```
fruit = ["kiwi", "ornage", "pear", "apple"]
print(fruit[1])  # Oops I wrote ornage
# Good thing we can replace it!
fruit[1] = "orange"
print(fruit)  # Same list object, but ornage is now orange
```

  - We can add items to the list without creating a new list using built-in **methods** *.append()* and *.insert().*

```
fruit.append("banana") # Adds an item to the end of the list
print(fruit) # List is ['kiwi', 'orange', 'pear', 'apple', 'banana']
fruit.insert(2, "pineapple") # Inserts an item at index 2, shifts items
print(fruit) # List now is ['kiwi', 'orange', 'pineapple', 'pear', 'apple', 'banana']
```

- We can also search for and remove items from a list.
  - Earlier when we replaced 'ornage' with 'orange', we had to manually look up the index for 'ornage'. Let's automate this using *.index().*

```
print(fruit.index('orange')) # Tells us the index of 'orange' is 1.
```

  - How do we cut out the middle man (us) here?

```
fruit[fruit.index('orange')] = 'clementine' # This is like writing fruit[1]
print(fruit.index('orange')) # Orange is no longer in list, throws error.
```

  - When we want to remove items, we don't need to know where they are, we can name them within the *.remove()* method.

```
fruit.remove('banana') # Banana is gone from the list.
fruit.pop(2) # We can also remove by index using the pop() method
print(fruit.pop(0)) # Pop outputs the value it removes. Potentially useful!
```

del fruit[2] is another option for deleting, but .pop() is more consistent syntactically

What’s in the Data Structure?

UNIVERSITY OF OXFORD

- We’ve seen one way of searching for an item and returning its index (but only works if it exists, otherwise error).
  - How do we ascertain whether an item is in the list or not? We can use ***in*** or ***not in*** which evaluates to True or False.

```
fruit = ["kiwi", "orange", "pear", "apple"]
print("cherry" in fruit)  # That’s false
pets = ("cat", "dog", "turtle")
print("bird" not in pets) # True because we’re using: not in.
print('i' in 'teamwork')  # These keywords work with strings too.
```

  - Particularly useful with IF statements.

```
fav = input("Who do you want to see?")
avail = [“Arnaud", “Lee", "Morales", "Carstens"]
if fav in avail:
      print("I’d like to see Dr.", fav)
else:
      print("whoever is available now")
```

Matrices: [[using, lists], [NumPy, Module]]

- If you needed to create a matrix of values, you could use a list of lists, the outer list delineating rows.

```
mat  = [[1,2],[3,4],[5,6]] # 3x2 matrix
```

$$matrix = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

  - This however, can lead to mistakes.

```
2*mat  # Attempting to x2 each element of matrix.
[[1,2],[3,4],[5,6],[1,2],[3,4],[5,6]] # Not what we wanted!
```

  - When a situation calls for matrices, use NumPy:

```
import numpy as np
nums = range(1,7) #1D number range from 1 to 6
pymat = np.reshape(nums, [3,2]) #NumPy 3x2 matrix

dmat = 2*pymat # x2 each element!
mmat = dmat*pymat # Matrix multiplication
np.transpose(mmat) # Transposes matrix
```

In your CLI, you will have to run:
pip install numpy
(But it will be worth it!)

Data Structures {'dictionaries': 1/2}

UNIVERSITY OF OXFORD

- One final built-in data structure to look is *dictionaries* or *dict*.
  - As with *lists* and *tuples*, *dictionaries* can store values of various types. Unlike these two, values are accessed directly using a **key**.

```
ext = {'Harrison Hamill': 1279, 'Carrie Ford': 1876, 'Mark Fisher': 1345}
# We know Carrie Ford (the key) we want her phone extension (the value)
print(ext['Carrie Ford'])  # Shows 1876
```

  - Although this won't be obvious on such small examples, getting a value using a key -a single step process- is much faster than looking through a list for a specific value.
    - This is why these types of structures are a fundamental part of aligning DNA sequences to the human genome (but that's a whole other class!...*)

*…Pipelines for NGS data specifically

Data Structures {'dictionaries': 2/2}

UNIVERSITY OF OXFORD

- In *dictionaries*, everything is mutable except for the individual keys themselves.
  - If Harrison Hamill gets a new extension, we can update that.

```
ext['Harrison Hamill'] = 1138
```

  - You can also use this approach to add a new entry.

```
ext['Anthony Baker'] = 3022 # Not previously in dictionary but now is.
```

  - Keys are also used for removing entries (using .pop() again).

```
ext.pop('Anthony Baker') # Previously in dictionary now isn't.
```

  - If we are not sure if a key exists, we can use ***in*** and ***not in***.

```
'Mark Fisher' in ext # It's true
```

  - Methods also exists to lookup all keys, values or key:value pairs*, but that often defeats the purpose of *dict*.

```
ext.keys()
ext.values()
```

```
ext.items()
```

*You should mostly only ever need (if at all) *.keys()*.

- Previously, we saw how to repetitively apply an action to an iterable data structure using FOR and WHILE loops.
  - Here's a new example with a list.

```
num = 0
for i in [5, 4, 20, 19, 1, 6]:
  if i % 2 == 0:
    num += i # If even we add.
  else:
    num -= i # If odd we substract.
print(num)
```

  - What if we want to apply an action in distinct parts of our code rather than sequentially in a loop? We will need to create a **function**.
    - We have seen several examples of Python built-in functions, such as *print()*, *bool()*, *input()*, *range()*,…etc. Now let's create our own!

def functions(1/2):

- Functions bear some resemblance to mathematical functions.
  - Let's turn the following function $f(x) = \frac{x^2}{3x}$ into a Python function.

```
# First we define the function.
def first_funct(x):
    num = x**2/(3*x)
    return num

#then we can execute it.
print(first_funct(20)) # 6.666…7
print(first_funct(100)) # 33.3333…6
```

The first line of any function we define will be: **def** function_name(parameter):

followed by a set of instructions. The output of the function is the value prefaced with the **return** keyword. Nothing is invoked after a return statement.

  - Once defined, a function can be invoked in any part of your code. It can even be invoked on itself (in fact, this is an integral part of something called recursion).

```
print(first_funct(first_funct(100))) # Output of one function is input of the next.
```

def functions(2/2):

- A **return** can appear in the last line of a block of code within a function and several **return** statements can coexist within a function given flow control structures such as IF statements.

```
def evod(x):
        if x % 2 == 0:
                return “even number“
        else:
                if x < 0:
                        return "negative odd number"
                else:
                        return "positive odd number"
                        print("This print will never be executed")
```

- A function can and often has more than one input or **parameter***.

```
def some_math(x, y, z):
        div = x/y
        return div + z
```

*And more than one comma-separated output

Reading a file

UNIVERSITY OF OXFORD

- To get some real work done with Python, we need to be able to read in large quantities of information from files.
  - To access a text file we use the function *open().* The function takes two parameters: a file location and a single character that signals whether we are '**r**'eading in or '**w**'riting to a file.

```
myfile = open(‘mytext.txt', 'r') # All strings. The 'r' signals 'read' mode
print(myfile.read()) # Prints the entire content of the file.
myfile.close()
```

  - The *.read()* method outputs the content of the entire file. Once this is done, the file is closed using the *.close()* method.
  - Typically, we don’t want to read a file all at once, but instead line by line. We typically use *.readline()* instead.

Reading each line of a file

UNIVERSITY OF OXFORD

- Each invocation of *.readline()* outputs the next line in a file.
  - If we are interested in the first 3 lines of a file, we can use a FOR loop.

```
myfile = open(‘mytext.txt','r')
for i in range(3): # Here we don’t use the variable explicitly, that’s allowed!
	print(myfile.readline())
myfile.close()
```

  - The file stream itself is an iterable data structure, which means we can also use the FOR loop directly on it.

```
myfile = open(‘mytext.txt','r')
for line in myfile:  # You don’t need to call the variable line for this to work.
	print(line)
myfile.close()
```

Writing to file

- If we are reading in big files and modifying them, we will want to keep the output in new files.
  - We use the same function as before but with the characters 'w' or 'a'. Both work the same when writing to a new file.

```
newfile = open('new_text.txt', 'w')
newfile.write("Hello World!\n")  # You now have a file with Hello World! in it.
newfile.write("second line…\n")  # The \n ensures you write to next line.
newfile.close()
```

  - Where 'w' and 'a' differ is when the file being written to already exists. While 'w' (write) will overwrite what already exists, 'a' (append) will start writing from the end of the file, preserving what was there before.

A simple concrete read and write example

- Reading and writing is often done in one steady stream.
  - We have a file with information we want to transform and output to another file. For example, from a list of tab-delimited names, we can generate e-mail addresses with help from methods like ***.replace()***, ***.split()*** and ***.join()***.

```
names = open('names.txt', 'r')
mail = open('emails.txt', 'a')

for line in names:
  line = line.replace("'", "0") # Any apostrophe in a name is replaced with 0
  elem = line.split() # We split a string around space and get a list.
  mail.write("_".join(elem)+ "@wow.ac.uk\n") # List back to string.
names.close()
mail.close()
```

Alternative read/write using *with* and *as*

- Another form of reading and writing to files uses keywords ***with*** and ***as*** to contain read/write operation in a self-closing loop.
  - This form does not require a .close() statement, the stream being closed when the block finishes its execution.

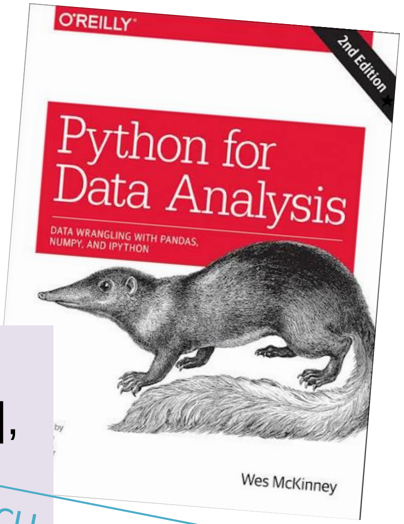
```
with open('emails.txt', 'a') as mail:
  with open('names.txt', 'r') as names:
    for line in names:
      line = line.replace(""", "0")
      elem = line.split() # .split() breaks each column into items.
      mail.write("_".join(elem)+ "@wow.ac.uk\n")
```

  - Note that our code is otherwise unchanged!

Check out modules **os, pathlib** and **glob** for more advanced file retrieval options

The Pandas Module (Series)

- We've seen the basic way of reading in files, but if we're working with organised data (e.g. table), is there something perhaps a little more advanced that can read structured data and summarise it? **Pandas**!
  - Pandas has become a cornerstone of data science with its addition of **Series** and **DataFrame** to Python. Let's first take a look at **Series**.




```
import pandas as pd
pd.Series([9.38,9.68,9.55,9.49], index=['2021','2020','2019','2018'], name='Average yearly UK temperatures')

2021    9.38
2020    9.68
2019    9.55
2018    9.49
Name: Average yearly UK temperatures, dtype: float64
```

In your CLI, you will have to run:
pip install pandas
(You're going to need it)

- If you add a dimension to **Series**, you end up with what looks like a table which we call here a **DataFrame**
  - While these structures can be simulated by lists of lists, as we saw with matrices (NumPy), packages bring a lot more functionality.

```
ukch=pd.DataFrame({'UK':[9.38,9.68,9.55,9.49],
                   'Chile':[11.57,12.71,12.36,12.63]},
                   index=['2021','2020','2019','2018'])

        UK   Chile
2021  9.38  11.57
2020  9.68  12.71
2019  9.55  12.36
2018  9.49  12.63

ukch.describe() # You can get several summary statistics this way
ukch.Chile.describe() #You can focus on specific columns by naming them.
```

- Let us now use **DataFrame** on real tabular data.
  - As an example, let’s load the (in)famous Titanic Dataset. For this we need the method **.read_csv()**, other methods exists for various kinds of tabular data.

```
titanic=pd.read_csv('https://www.well.ox.ac.uk/~miossec/courses/GMS2022/Titanic.csv')
# Let’s first look at a few rows of the data.
titanic.head()
# Index seems redundant, let’s PassengerId as index.
titanic=titanic.set_index("PassengerId") # Don’t forget to re-assign to DataFrame.
# Let’s query the data using .loc and conditional arguments in Pandas’ style.
titanic.loc[(titanic.Survived == 1) & (titanic.Fare <= 10)] # & eq. to AND, | eq. to OR
# Let’s look for passengers of 2nd and 3rd class, but only show their name and age.
pnames=titanic.loc[titanic.Pclass.isin([2,3])][['Name', 'Age']]
# Maybe we can save that reduced table to its own .csv?
pnames.to_csv("names_subset.csv")
```

Part II challenges

- Here are a few practical exercises you can now tackle.
  - Write a function that takes as input two short DNA sequences such as 'ATGGTCA' and checks whether they are complements of each other.
  - From a file (you create manually) containing a column of temperatures in degrees Celsius, generate a file with three columns for degrees Celsius, Fahrenheit and Kelvin (hint: +"\t"+ to add tab).
- An inescapable part of programming is encountering errors.
  - Rather than do everything to avoid them, it’s a good idea to familiarise oneself with them early on. Write bits of code that leads to as many of the following errors (without looking them up at first):
    - IOError, IndexError, KeyError, NameError, TypeError, ValueError, ZeroDivisionError.

Part II Challenge: Wordle?

UNIVERSITY OF OXFORD

- Create a simple text desktop-based version of the newly popular internet word game Wordle: (https://www.nytimes.com/games/wordle/index.html)
  - Your program must take a list of 5-letter words (any language you chose) from a text file and select one at random.
  - Prompt the user to try to guess the word 6 times. Showing which letters they got right, which they got wrong and which they just got in the wrong place.
  - Keep it simple, make it all text based, using input() to prompt the guesses. Don't forget to apply **divide** and **conquer** approach.

Thanks for listening!
Happy scripting!

Extra: Stuck on a small problem? Google it

- You can't always guess what the best way of doing something is. Every single programmer, from total beginner to total pro, needs some help from the Python community.
  - Want to keep learning or review more concepts? A good place to start is Python's official page: https://www.python.org/doc/
  - Have a more in-depth question? There's a good chance someone on stackoverflow has had the same question as you (or similar) and someone has offered an answer: https://stackoverflow.com/questions/tagged/python
    - If you google your question, you will probably end up on stackoverflow.

Stackoverflow is useful for other languages as well!

Extra: Python packages

UNIVERSITY OF OXFORD

- Once you're comfortable enough with base Python, you can start exploring packages! We've explored a few already.
- You can use them to do some pretty cool things in Python without having to reinvent the wheel:
  - Would you like to make graphs? `import matplotlib, seaborn`
  - Matrix calculation and data analysis? `import numpy, pandas`
  - Want to make a graphical user interface? `import Tkinter`
  - Want to make some machine learning, maybe artificial or convolutional neural networks without having to create the whole infrastructure from scratch?

    `import scikit-learn, tensorflow, keras, pytorch`