Introduction to Python

Part II: 19/10/2020

Matthieu Miossec, PhD
Duncan Parkes, PhD

Part I challenge, a few solutions

- Did you manage?
  - Write a script that takes the numbers 1 through 100, but prints fizz if said number is a multiple of 3, buzz if a multiple of 5 and fizzbuzz if a multiple of 5 and 3. Here are a few solutions…

Remember, 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 independent IF statements here.

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

A non-empty String is True. Faster than writing len(out) !=0

Data Structures

- On Friday, we looked at a number of in-built data types (i.e. String, integer, float boolean).
  - Python also has what are called **data structures**, more specifically tuples, 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
```

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

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

  - Python even allows for them to store different data types within the same structure.

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

  - They may also contain themselves and each other!

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

You can use type() on data structures as well

Data Structures –tuples– 1/2

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

```
weather = ("sun", "clouds", "rain", "snow")
print(weather[1]) #clouds. We can access an element of the tuple.
#but we can’t replace or remove it.
weather[1] = "mist" #This will cause an error!
```

  - Also as with Strings, we can get a slice of a tuple…

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

  - …determine length, concatenate tuples or multiply them.

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

Data Structures –tuples within tuples–

- As mentioned earlier, tuples can contain other data structures.
  - If a data structure exists within another, we can still access its elements in the following manner.

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

print(some_strings[2][1]) #14
print(some_strings[0][0]) #sun
print(some_strings[1:3][1]) #('rain', 22).
```

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

```
some_slice = some_strings[1:3]
print(some_slice) #("clouds", 16.2), ("rain", 14)
print(some_slice[1]) #('rain', 22). Same result in more steps.
```

Which tuple leads to a False when added to *bool()*?

Data Structures –lists– 1/2

- We’ve seen that both Strings and tuples are immutable. What does a **mutable** data structure look like?
  - For this we look at **lists**.

```
fruit = ["kiwi", "ornage", "pear", "apple"]
print(fruit[1]) #ornage (oops). We can access an element of the list.
#and this time we can replace it!
fruit[1] = "orange"
print(fruit) #same list, but ornage is now orange
```

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

```
fruit.append("banana") #Adding an element to the end of the list
print(fruit) #List is ['kiwi', 'orange', 'pear', 'apple', 'banana']
fruit.insert(2, "pineapple")
print(fruit) #List now is ['kiwi', 'orange', 'pineapple', 'pear', 'apple', 'banana']
```

Data Structures –lists– 2/2

- We can also search for or remove elements 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().* replace ‘orange’ with ‘clementine’.

```
print(fruit.index('orange')) #Tells us the index is 2
```

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

```
fruit[fruit.index('orange')] = 'clementine‘ #this is like writing fruit[2]
#orange is no longer in the list
print(fruit.index('orange')) #throws an error!
```

  - When we want to remove elements, we don’t need to know where they are, we can name them using *.remove()*

```
print(fruit.remove('banana')) #banana is gone from the list.
del fruit[0] #we can also remove by index using del as a prefix.
print(fruit) #unlike .remove(), you can’t put del in a print statement.
```

What’s in the Data Structure?

- We have seen one way of searching for a specific element in a list and returning its index (if it exists).
  - What if we are only interested in whether an element is in the list or not? We can use *in* or *not in* which give us True or False values.

```
fruit = ["kiwi", "orange", "pear", "apple"]
print("cherry" in fruit) #False
animals = (cat, dog, bear)
print("bird" not in animals) #True. Notice we’re using: not in.
print('i' in 'teamwork')  #It works with Strings too.
```

  - This can be particularly useful with IF statements.

```
fav_doctor = "Morales"
available_doctors = ["Henderson", "Banner", "Morales", "Rath"]
if fav_doctor in available_doctors:
        print("I’d like to see Dr.", fav_doctor)
else:
        print("whoever is available now")
```

Data Structures –dictionaries– 1/2

- The final data structure we'll look at is dictionaries.
  - As with lists and tuples, dictionaries can store elements of various types. Unlike the other two, elements are accessed directly using a unique key.

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

  - Although this might not be obvious on such small examples, getting a value using a key, a one step process, is much faster than looking through a list for a specific value.
    - In fact, these types of structures are partly used to align DNA sequence reads to the human genome (but that's a whole other course!)

Data Structures –dictionaries– 2/2

- In dictionaries, everything is mutable except for the keys themselves.
  - If Harrison Hamill changes extension, we can update that.

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

  - You can use the exact same approach for adding a new entry.

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

  - Keys are also used for removing whole items.

```
del office_number['Anthony Baker'] #previously in dictionary now isn’t.
```

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

```
'Mark Fisher' in office_number #True
```

  - Methods also exists to lookup all keys, values or key:value pairs*.

```
office_number.keys()
office_numbers.values()
```

```
office_number.items()
```

*You should mostly only ever need *.keys()*.

Beyond flow control: Functions

- Last week, 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.

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

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

Functions 1/2

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

```
#first we define the function.
def first_function(x):
    number = x**2/(3*x)
    return number

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

The first line of any function we define will be: **def** name_of_function(input):

This is then followed by a set of instructions. The output of the function is preceded by **return** and it is the last line that is invoked.

  - Once defined, a function can be invoked in any part of your code. It can even be invoked on itself!

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

Functions 2/2

- Although nothing can be executed after a **return**, it doesn’t need to be in the last line of a function and there can be several **return** statements in a function because of IF statements.

```
def even_odd(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 divr + z
```

[*]And more than one comma-separated output

Reading a file

- In order to get any real work done with Python, we need to be able to read in information from files.
  - In Python, to access a human-readable (i.e. text) file we use the function *open().* The function takes two parameters: the location of the file and a single character that signals whether we are reading the file or writing to the file.

```
my_file = open('some_text.txt', 'r') #all in quotes. The 'r' means were 'reading'
print(my_file.read()) #prints the entire content of the file.
my_file.close()
```

  - The *.read()* method outputs the content of the entire file. Once this is done, the file is closed using *.close()*.
  - Typically, a file is too big to be read all at once. In this case, we favour *.readline()*

Reading each line of a file

- 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.

```
my_file = open('some_text.txt','r')
for i in range(3): #here we don’t use the variable explicitely.
        print(my_file.readline())
my_file.close()
```

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

```
my_file = open('some_text.txt','r')
for line in my_file: #here we don’t explicitly use the i variable.
        print(line)
my_file.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.

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

  - Where 'w' and 'a' make a difference 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 the methods *.replace()*, *.split()* and *.join()*.

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

for line in name_file:
  line = line.replace("'", "0") #any apostrophe in a name is replaced with 0
  name = line.split() # .split() breaks each column into elements.
  email_file.write("_".join(name)+ "@wow.ac.uk\n")
name_file.close()
email_file.close()
```

Part II challenge

- Here are a few practical exercises you can now tackle.
  - Write a function that takes as input a short DNA sequence such as 'ATGGTCA‘ and prints the String 'This DNA sequence is *x* base pair long', where *x* is the length of your input sequence.
  - From a file (you can create manually) containing a column of temperatures in degrees Celsius, create 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):
    - IOError, IndexError, KeyError, NameError, TypeError, ValueError, ZeroDivisionError.

Thanks for listening!
Happy scripting!