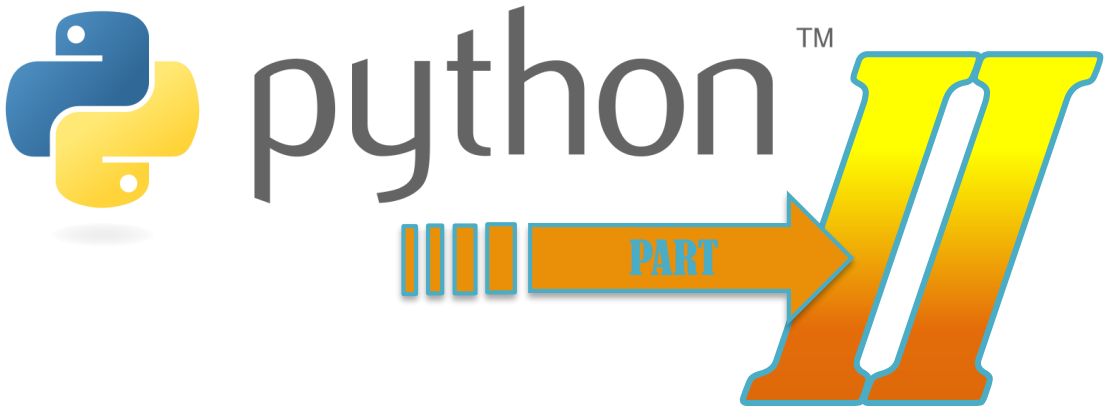
Getting started with



Matthieu Miossec, PhD
Duncan Parkes, PhD

"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."

Did you manage? Here are a few 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 independent IF statements here.

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

var+= other_var
is a shorthand for:
var = var + other_var

A non-empty String is **True**.
Less operations than len(out) !=0

- Last week, we looked at a number of built-in data types (i.e. 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 even allows for them to store distinct 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
```

- 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 in the tuple.
#We can't replace or remove it.
weather[1] = "mist" #This will cause an error!
```

  - 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
```

- As mentioned previously, tuples can contain other data structures.
  - If a data structure exists within another, we can still access its elements 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.
```

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

```
fruit = ["kiwi", "ornage", "pear", "apple"]
print(fruit[1]) #ornage (oops).
#Good thing 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 and 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().*

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

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

```
fruit[fruit.index('orange')] = 'clementine‘ #this is like writing fruit[1]
#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()*

```
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)
```

You can also use .pop(0) to remove the element in index 0

- 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 a **True** or **False** value.

```
fruit = ["kiwi", "orange", "pear", "apple"]
print("cherry" in fruit) #False
pets = ("cat", "dog", "turtle")
print("bird" not in pets) #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_doc = input("Who do you want to see?")
docs_avail = ["Henderson", "Banner", "Morales", "Rath"]
if fav_doc in docs_avail:
        print("I'd like to see Dr.", fav_doctor)
else:
        print("whoever is available now")
```

- 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**.

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

  - Although this might not 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.
    - In fact, these types of structures are a fundamental part of aligning DNA sequences 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 gets a new extension, we can update that.

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

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

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

  - Keys are also used for removing whole items.

```
del off_num['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 off_num #True
```

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

```
off_num.keys()
off_nums.values()
```

```
off_num.items()
```

You can also use .pop(0) to remove the element in index 0

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

- 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.

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

  - What if we want to apply an action in distinct parts of our 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 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_function(x):
    num = x**2/(3*x)
    return num

#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.
```

def 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 div + z
```

*And more than one comma-separated output

- In order to get any real work done with Python, we need to be able to read in large quantities of 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 **r**eading the file or **w**riting to the file.

```
myfile = open('mytext.txt', 'r') #all in quotes. 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 *.close()*.
  - Typically, a file is too big to be read all at once. We typically favour using *.readline()* instead.

- 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.
        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: #here we don't explicitly use the i variable.
        print(line)
myfile.close()
```

- 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' are different 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 written before.

- 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()*.**

```
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() # .split() breaks each column into elements.
  mail.write("_".join(elem)+ "@wow.ac.uk\n")
names.close()
mail.close()
```

- Another form of reading and writing to files involves using the keywords ***with*** and ***as***.
  - 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 elements.
      mail.write("_".join(elem)+ "@wow.ac.uk\n")
```

  - Note that our code is otherwise unchanged!

- 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.

Thanks for listening!
Happy scripting!