Getting started with

python™ PART I

Matthieu Miossec, PhD (@RealMattJM)

Bioinformatics Core

https://www.well.ox.ac.uk/people/matthieu-miossec

28/10/2021

- Python is a **high-level programming language** originally developed by Guido Van Rossum in 1991.

  - The last major revision of the language in 2008 led to Python 3[*], but the language is continually evolving (latest version 3.10.0 released early October).

- To understand the driving philosophy behind Python, we can write our first line of code.

```
import this
```

 - The resulting 19 statements are what Tim Peters calls the **Zen of Python**.

High-level programming language?

- A high-level programming language is one with a high level of **abstraction** from the details of a computer system.
    - Closer to natural language (typically English).

Why learn Python?

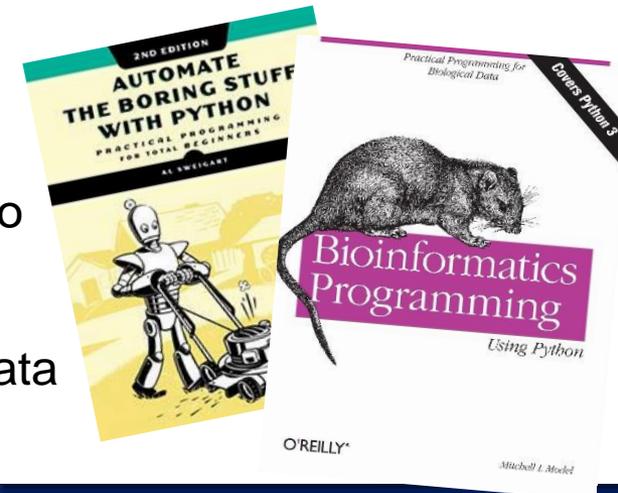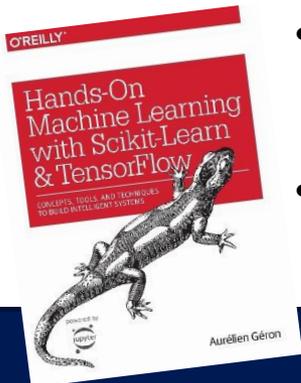| Oct 2021	Change		Programming Language	Ratings
1	▲		Python	11.27%
2	▼		C	11.16%
3	▼		Java	10.46%
4			C++	7.50%
5			C#	5.26%
6		VB	Visual Basic	5.24%
7		JS	JavaScript	2.19%
8	▲	SQL	SQL	2.17%
9	▼	php	PHP	2.10%
10	▲	ASM	Assembly language	2.06%

(source: TIOBE programming community index as of October 2021)

- One of the top programming languages right now. With applications in a variety of industries.
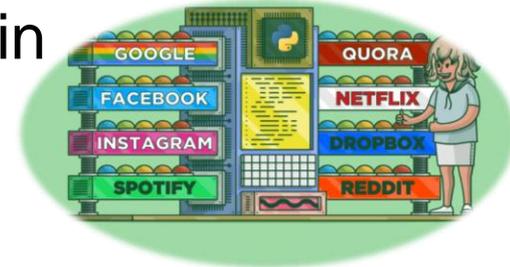

(see: https://realpython.com/world-class-companies-using-python/ for some examples)

- Doesn't require writing an entire class just to print "Hello World!" to screen (unlike Java).

- A popular choice for scientific computing, data science and machine learning.

Hands-On Machine Learning with Scikit-Learn & TensorFlow — Aurélien Géron

AUTOMATE THE BORING STUFF WITH PYTHON

Bioinformatics Programming Using Python

- Once you have Python installed[*], you can start writing and executing Python code directly into the command line terminal.

  - **<u>Interactive mode</u>**: In a terminal, type either *python* or *python3*. This will start the **Python prompt/shell** in which you can type individual commands:
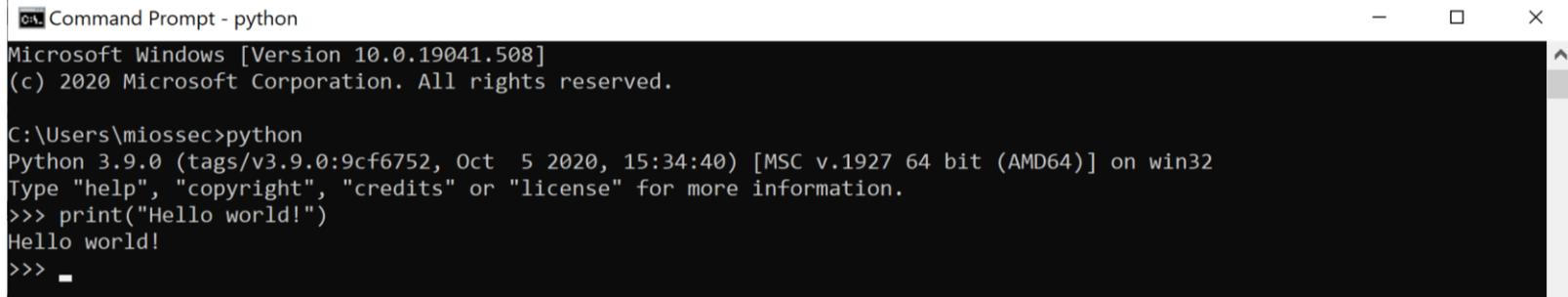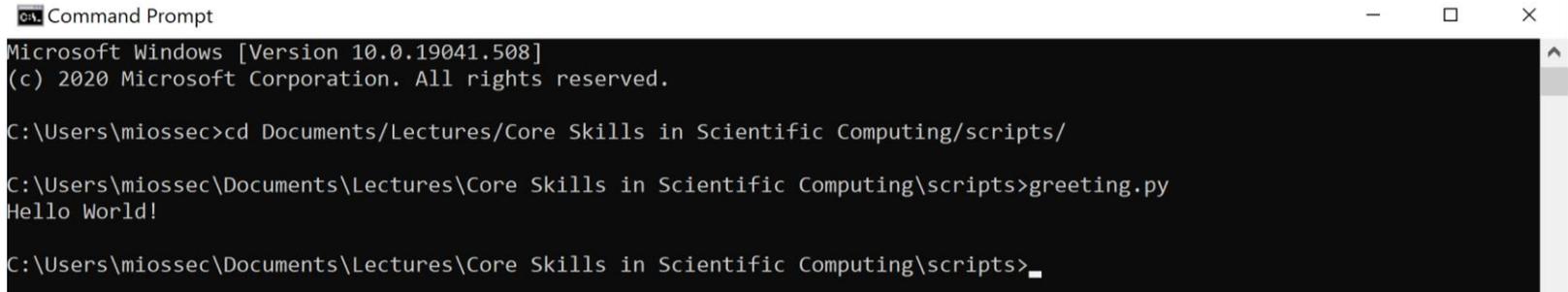

  - When you're done, type *quit()* to exit.

- Interactive mode is a good place to start, but to execute full blocks of code, you will need to write them in a python script file (*.py* extension).

  - **Script mode**: You can use a simple text editor or an Integrated Development Environment (IDE) to write your code and then execute the resulting **script** from the terminal.

  - If you are using an IDE, you can both write <u>and execute</u> your code within that environment!

- An **integrated development environment** (IDE) provides an interface with which to write code, a kernel to execute said code and a combination of useful tools to assist in these tasks.

    - Recurring features include debugging, auto-complete, linting (checking for potential sources of errors as you write code), file-management and version control.

- Several IDEs are compatible with (e.g. Atom, Sublime, Eclipse) or made for Python (e.g. PyCharm, Spyder, PyDev and JupyterLab).

    - A good IDE to start with if you're running Python on your own machine is the IDLE[*], created **in** Python **for** Python.

- For the purpose of this course, we will be working with an online IDE called **Repl.it**.

Let's start coding!

- By the end of today's session you should have all the necessary tools to implement the following script:

  - 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.

- As we cover various aspects of Python today, think about what might be useful for these various aspects of the problem:

  - Identifying multiples of 3 or 5 or both.

  - Running through a list of numbers.

  - Performing an action when a specific condition is fulfilled

- Python has several numeric types, we will start by looking at *integers* (*int*).

  - We can use several familiar **operators** to join **values** into **expressions** which **evaluate** to a new **value**:

    ```
    7 + 3 #Evaluates to 10.
    7 - 3 # 4
    ```

    ```
    7 * 3  #Multiplication. 21
    7 / 3  #Division. 2.3333333333333335
    ```

  - The above division operation gives us a floating-point number (*float*). If we want to ignore the fractional part of a division, we can replace it with //, or if we want the modulus/remainder %.

    ```
    7 // 3  # 2
    ```

    ```
    7 % 3  # 1
    ```

  - We can also raise values to a given power.

    ```
    7 ** 3  # 343
    ```

- We can use the same operators to work with floats or a combination of integers and floats.

    - An expression that includes a float will evaluate to a float, even if the fractional part is 0.

        ```
        5.2 * 5 #Evaluates to 26.0 instead of 26
        type(5.2 * 5)  #type() shows <class 'float'>
        ```

    - If we want an object of type *int* (integer) we can use **casting** which converts a data type into another. Here with the function *int()*.

        ```
        int(5.2 * 5)
        ```

    - Casting **only removes the fractional part**, it **does not** round numbers to the nearest integer.

        ```
        int(5.2 * 8) #Evaluates to 41 with the original float value being 41.6
        round(5.2 * 8) #Use round() instead for rounding up to 42.
        ```

- We've already seen one example of the *print()* function.

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

  - The statement contained in the double quotes is a String (i.e. string of characters). In Python, we can also use single quotes to denote a String.

    ```
    print('Hello World!')
    ```

  - If your String itself contains single quotes or an apostrophe, you can either escape the character or use double quotes.

    ```
    print('Season\'s greetings')
    print("My first message said 'Hello World!'")
    ```

  - And vice versa.

    ```
    print('My first message said "Hello World!"')
    ```

- In Python, we can input several separate values into *print()*, the result of which will be a single output message.

```
print("Hello",'wonderful',"World!") #Notice the spaces between each element.
```

- This allows us to use different data types and expressions.

```
print("The year is", 2019+1)
```

- …or booleans (more on these soon!).

```
print(2+2<4) #This will evaluate to False, with a capital F
```

- But most importantly, it is useful for inputting **variables** into a message. Let's create a new variable <u>during execution</u> using *input()*.

```
adjective = input("Your input here:")
print("Hello", adjective, "World")
```

- A **variable** provides a way to label and access an object without needing to know where it is stored in the computer's memory. Here's one example of assigning a value to a variable: `word = "bird"`

  - In most programming languages, we can assign the value of a variable to another variable.

    ```
    num = 5
    other_num = 3
    num = other_num
    ```
    `print(num)  #outputs 3`

  - In this case, **only** the value is assigned. Any changes to one variable won't affect the other afterwards.

    ```
    other_num = 7
    print(num) #Still 3
    ```

- A String object is actually a subsettable string of characters.

  - We can access specific elements of a String stored in a variable using an **index** between square brackets.

    ```
    greet = "Hello World!"
    print(len(greeting)) #What is the string's length? Find out using len()

    print(greet[0]) #H. first element of the String.
    print(greet[11]) #!. Last element of the String.
    print(greet[0:5]) #The word Hello.
    ```

  - The final **slice** shows elements 0 through 4. Elements are **indexed from 0** and the index after the colon is **not included***.

  - Strings are **immutable**, you cannot change individual characters…except by replacing the entire String.

    ```
    greeting[11] = "?" #This will cause an error!
    ```

- There exists a shorter way of referencing the start and end of a String.

```
greeting = "Hello World!"
print(greeting[:5]) #Hello. Absence of index implies the start index.
print(greeting[6:]) #World! Likewise, end index is implied.
#What happens if you type the following? Is it useful?
print(greeting[:])
```

  - Although this might not seem useful for now, it is also possible to use negative indices which refer to elements from the end of the String.

```
print(greeting[-1]) #! Last element.
print(greeting[-1:-3]) #This doesn't work.
print(greeting[-3:-1]) #ld. This does.
print(greeting[9:-1]) #ld. And this.
```

H	e	l	l	o		W	o	r	l	d	!

0	1	2	3	4	5	6	7	8	9	10	11	(12)
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	
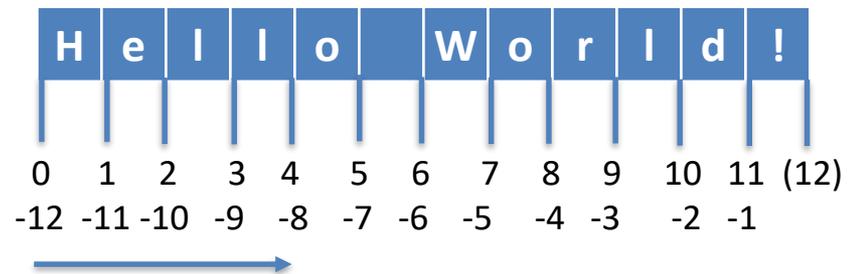
- Applying operators directly to values is of limited use.

  - Luckily, we can apply them to variables and assign the evaluation of the resulting expression to a new variable.

    ```
    new_num = (num + other_num + 10) / 2
    print(new_num)
    ```

  - When working with Strings, the + symbol takes on new meaning: We can use it to join (**concatenate**) Strings together.

    ```
    greet = "Hello"
    place = "World!"
    msg = greet+" "+place
    print(msg)
    ```

    The * symbol (followed by an integer) can also be used to repeat a String a given number of times.

    ```
    print(greet[0:2]*3) #HeHeHe
    ```

  - You <u>cannot</u> concatenate a String and another object unless you cast the latter as a String first using *str()*.

    ```
    print("Hello it's "+2020) #what does the error code say?
    ```

- An important data type we have not explored yet is boolean (*bool*).

  - Booleans have only two possible values: <u>T</u>rue or <u>F</u>alse[*].

  - In Python, every data type inherently has a True or False value.

    - Little exercise: Can you figure out what is True or False for *int* and *String* using *bool()*?

      ```
      print(bool(4)) #True
      #Try a few notable integers…
      print(bool("Hello")) #True
      #What could possibly cause False for a String?
      ```

  - Despite their simplicity, booleans are very powerful once we introduce **comparison operators** and **flow control** (IF statements, FOR/WHILE loops…).

- To compare two values or expressions, we use comparison operators.

  - As with mathematical operators, many are straightforward.

    Num_a, num_b, word = 4, 10, "hello" #This is a neat Python trick…
    print(num_a < num_b) #Less than. True
    print(num_a > num_b) #Greater than. False
    print(num_b >= num_a) #Greater or equal to. True
    print(word <= "Hello") #case matters. False

  - Another common source of error is == which stands for equals to. Not to be confused with = which is for variable assignment.

    print(num_b == 10) #Equals to. True

  - To evaluate two values as not equal, we use !=

    print(num_a != 10) #Not equal. True

  - All these comparison operations lead to True or False statements.

- Comparison expressions and boolean variables can be further joined together using logical connectors such as **and**…

```
num = int(input("Is this number even and bigger than 3?"))
print(num%2==0 and num>3) #True if both statements true. False otherwise.
```

- …**or**…(this is an **<u>inclusive or</u>**, if both statements are True, evaluates to True)

```
num = int(input("Is this number negative or even"))
print(num<0 or num%2==0) #True if both or at least one statement true.
```

- …**not**…

```
some_value = False
print(not some_value) #Is this value False? True if False. False if True.
```

- Boolean still might feel of limited utility at this point. In fact we won't achieve anything of much use up to this point. This is where **flow control** comes in, starting with **IF statements**.

- So far a lot of what we've looked at can fit into single line statements. However, an **IF statement** takes up a few lines.

```
num = int(input("Is this number even?"))

if num % 2 == 0 : #when True, the block of code below is executed.
    print(num, "is even")
else:                   #Otherwise this block of code is.
    print(num, "is not even")
```

- Here is the general syntax:

```
if(condition):
        some action.
        some more action.
else:
        some other action
        some more other action
```
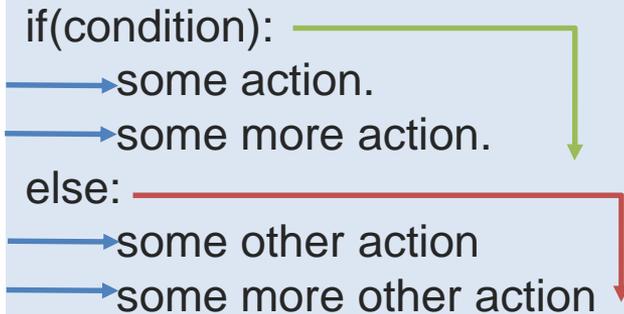
- Notice the indentation. This is how you separate blocks of code in Python. Only what is indented after IF will be executed if the condition evaluates to True. If False, only the block after else will be executed.

- What if we want to evaluate more than one conditional statement? For this, we use **ELIF** (a contraction of else if).

```
num = int(input("Is this number negative, 0 or positive?"))

if num == 0 :
    print("is", num)
elif num > 0 :
    print(num, "is positive")
else:
    print(num, "is negative")
print("and that's final.")
```

- Notice this final print statement appears to be aligned with the if structure, meaning that it is outside of it and will be executed regardless.

- An IF statement can contain any number of ELIF statements[*].

- Sometimes, more than one condition could evaluate to True. Say we want to know if a number is 4, even or odd.

```
num = int(input("Is this number 4, even or odd?"))

if num == 4 :  #only the indented block below is executed if number is 4
    print("is", num)
elif num % 2  == 0 : #this is never checked if previous condition is True.
    print(num, "is even")
else:               #this won't be checked either.
    print(num, "is odd")
```

- The number 4 is even. Two statements therefore could evaluate to True. So what happens in this case?

- In Python, as in other programming languages, <u>only the first True statement is evaluated</u>.

- One of the benefits of programming is that it allows us to perform repetitive actions in an automated fashion.

  - One way to apply an action repeatedly in Python, is by using a FOR loop. As with the IF statement, the block to be executed repeatedly is indented.

```
for i in range(100):
    print(i+1) #Why +1?
```

  - This also works for a String.

```
#spelling my name on the phone
lname = "Miossec"
print("yes, it's...")
for letter in lname:
    print(letter)
    print("...") #got to pause, right?
```

```
for variable in iterable data structure:
    some action
    some action on current variable
    some other action
```

The first element of the iterable data structure is assigned to the variable and some actions are performed. The second element is assigned to the variable…and so on until the last element.

- A WHILE loop also repeats a block of code, but does this **while** a certain condition is True.

  - Our previous counting to 100 example with a WHILE loop.

  ```
  i = 0 #starting value for variable i
  while i <= 100:
    print(i)
    i = i + 1 #don't forget this!!
  print("done!")
  ```

  - When writing WHILE loops, be very careful or else your loop could be infinite!

  ```
  #Hit ctrl-C when you realise the mistake!
  i = 0
  while i <= 100:
    print(i) #this keeps given us 0!
  print("done!") #This statement will never be reached.
  ```

- A common interview question* that uses some of the things we've covered today:

  - 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.

- To approach this problem, as with any problem in programming, **divide and conquer**.

  - Break the big problem into smaller manageable sub-problems and built up to the ultimate solution.

    - For example, before writing a program that prints out fizz, buzz and fizzbuzz, write a program that prints numbers 1 through 100 and build on that

- …step 1 of writing good code is writing code that works.

- You can always revisit your code and make incremental improvements -> optimising.

Thanks for listening, see you next week for

- What if we want to include both types of quote in the String?

    - Python allows for triple quotes! (single/double quotes x 3)

    ```
    print("""Here's a "great" example""")
    ```

   - Triple quotes can interpret spaces and new lines. This is a fairly unique feature to Python.

    ```
    print("""Here is a fragment of DNA:
        …A T G A A C…
         | | |  | | |
        …T A C T T G…
          """)
    ```

    - For everyday programming tasks, single or double quotes are usually adequate. We can always skip to a new line using \n.

    ```
    print("Hello World! \nHow's it going?")
    ```

- A FOR loop will typically iterate through each element of an iterable data structure.

  - That is, unless there is a reason to 'break out of the loop early'. This is done using **break** and involves combining a FOR loop with an **IF statement**.

```
prod = 1
for num in [2, 100, 0, 4.3, 6]: #This is a list, more on these in part II
    if(num==0):
        prod = 0 #Notice the double-indentation.
        break          #Multiplying anything by 0 gives 0 again, exit loop
    else:
        prod*= num
print(prod)
```
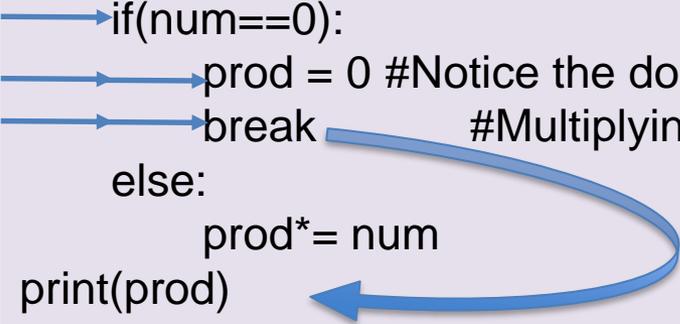
  - It is also possible to simply skip to the next iteration using the keyword **continue**.

Extra: Why is it called Python?
Is Guido Van Rossum really into snakes?

**wellcome
centre
human
genetics**

UNIVERSITY OF OXFORD

Eric Idle ✔ @EricIdle · Jan 29

I wish I understood what this was. But it's kind of nice. Let's not forget the origin of Spam on the internet.

💬 11   🔁 37   ♡ 424   ⬆

Guido van Rossum ✔
@gvanrossum

Replying to @EricIdle and @KansasGrant

Hi @EricIdle, author of said "Python" software here. It's open source, which means it's free. It's been 31 years since I chose the name and I'm still a fan of your work! It's "very nice". Many folks working with me in the early days shared my appreciation.

12:22 AM · Jan 30, 2021 · Twitter Web App

164 Retweets   **89** Quote Tweets   **1,364** Likes