Introduction to Python

Part I: 25/01/2021

(Part II: 27/01/2021)

Matthieu Miossec, PhD
Bioinformatics Core

- 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 3.9 released in November).

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

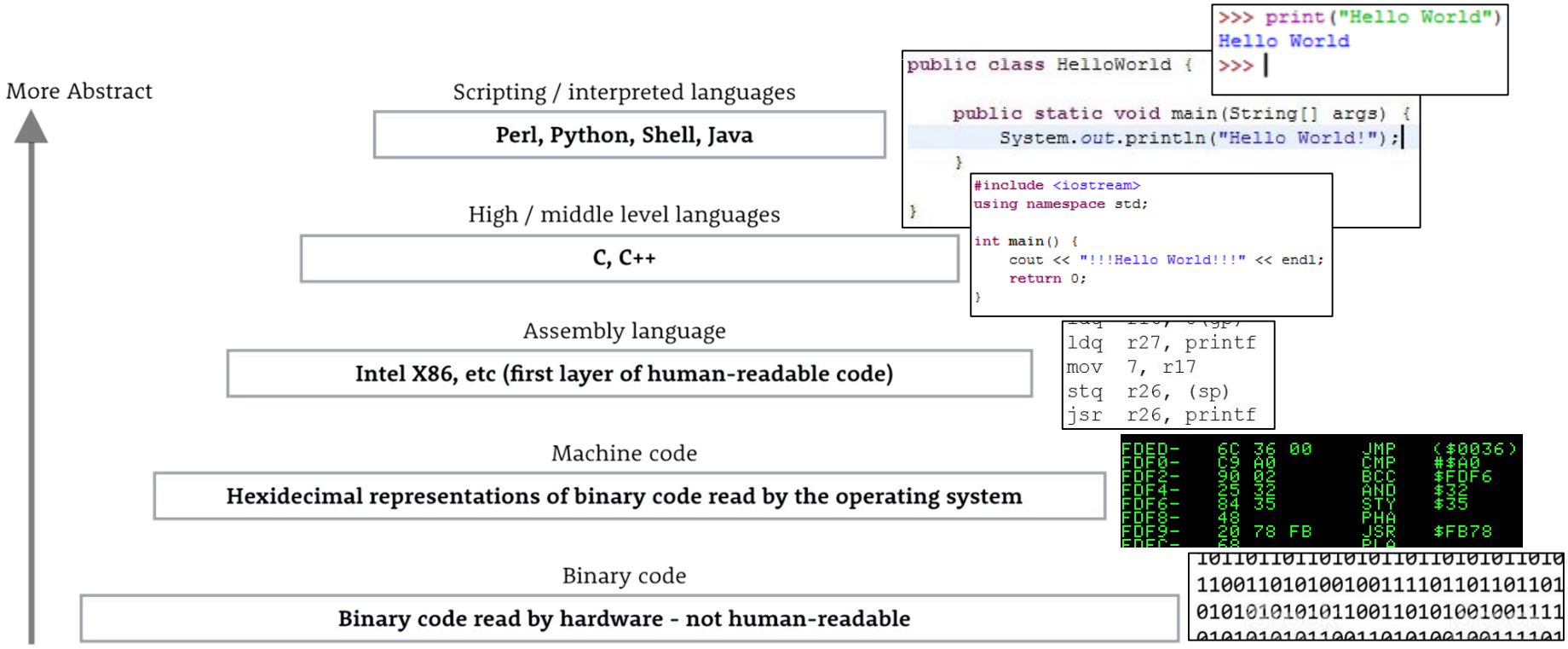
```
import this
```

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

*Please avoid Python 2, it has officially been discontinued.

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?

BIG DATA INSTITUTE

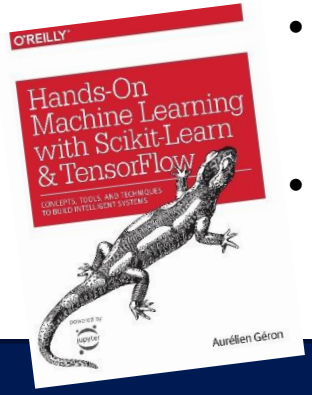
UNIVERSITY OF OXFORD

Oct 2020	Programming Language	Ratings	Change
1	C	16.95%	+0.77%
2	Java	12.56%	-4.32%
3	Python	11.28%	+2.19%
4	C++	6.94%	+0.71%
5	C#	4.16%	+0.30%
6	Visual Basic	3.97%	+0.23%
7	JavaScript	2.14%	+0.06%
8	PHP	2.09%	+0.18%
9	R	1.99%	+0.73%
10	SQL	1.57%	-0.37%

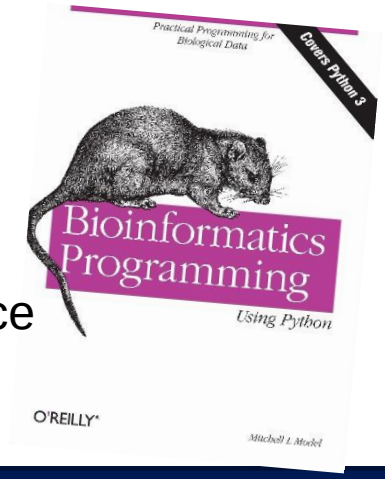
(source: TIOBE programming community index as of October 2020)

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)

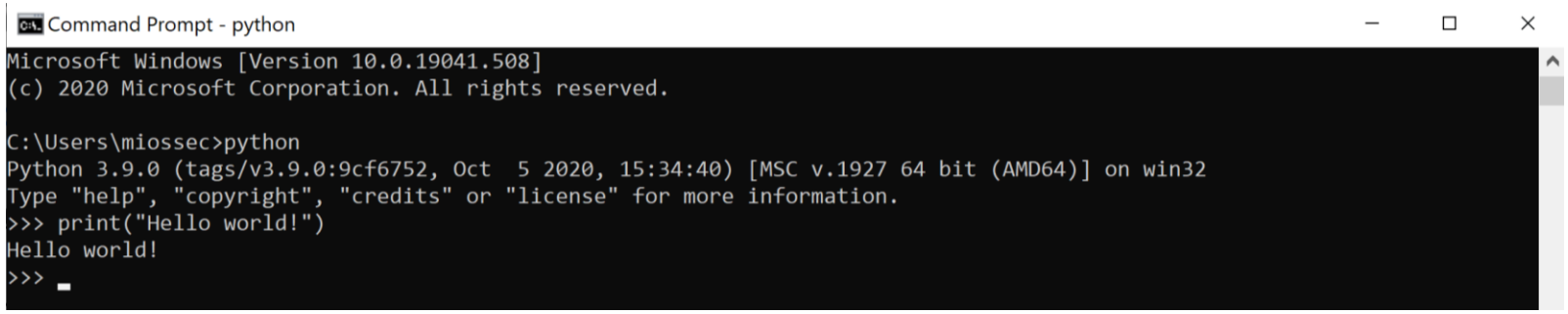


- Doesn't require writing several lines of code just to print "Hello World!" to screen (unlike Java).
- A popular choice for scientific computing, data science and machine learning.



How to use Python? –Interactive mode–

- If you have Python installed on your machine*, you can start writing and executing Python code directly into the command line terminal.
  - **Interactive mode**: open a terminal and type either *python* or *python3*. This will start the **Python prompt/shell** in which you can type individual commands:

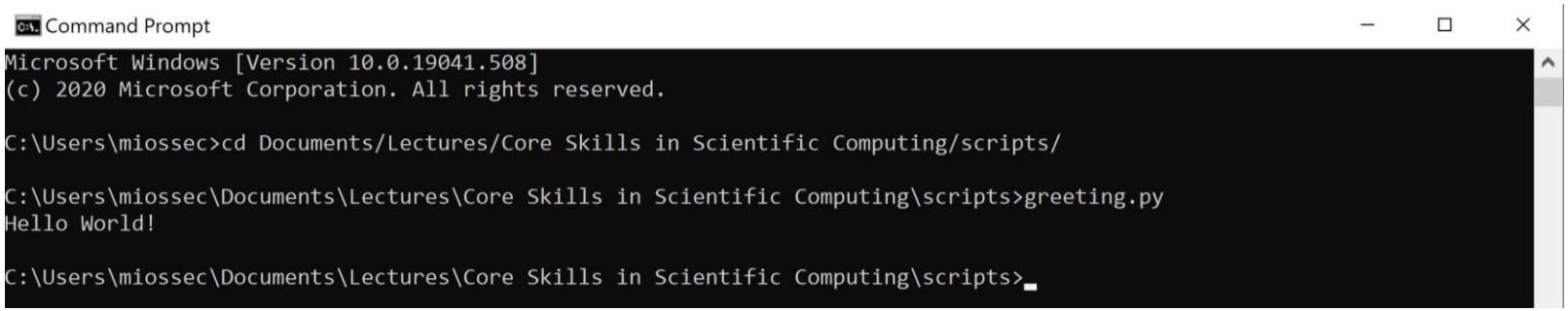


  - When you're done, type *quit()* to exit.

*We highly recommend you do this after the class if you haven't already.

How to use Python? –script mode–

- Interactive mode is a good place to start, but to execute full blocks of code, you will need to write them into a python script file (*.py* extension).
  - **Script mode**: You can use a simple text editor or an IDE to write your code and then execute the resulting **script** from the command line.



  - If you are using an IDE, you can both write and execute code within that environment!

How to use Python? –IDE–

- 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.

*The extra L is for learning.

- 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 #This is a comment.
#The above expression 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) #The function type() will show <class 'float'>
```

  - If we want to get 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)
```

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

```
int(5.2 * 8) #Evaluates to 41 when the original float value is 41.6
round(5.2*8) #Use the function round() instead for 42.
```

type() tells you what type of value you're working with.

- 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.

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

  - If your String 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!"')
```

Triple Quotes in Python

- 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?")
```

More on *Print()*

- 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 on the fly 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.

```
a_number = 5
another_number = 3
a_number = another_number
print(a_number) #3
```

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

```
another_number = 7
print(a_number) #Still 3
```

Manipulating a String (of characters) variable 1/2

- A String object is actually a string of characters.
  - We can access specific elements of a String stored in a variable using an index between square brackets.

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

print(greeting[0]) #H. first element of the String. First index is 0 not 1.
print(greeting[11]) #!. Last element of the String.
print(greeting[0:5]) #The word Hello.
```

  - The above **slice** shows elements 0 through 4. Elements are **indexed from 0** and the index after the colon is **not included***.
  - Note that strings are **immutable**, you cannot change individual characters…except by replacing the whole string completely.

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

*This is the source of many errors, so try to keep that in mind.

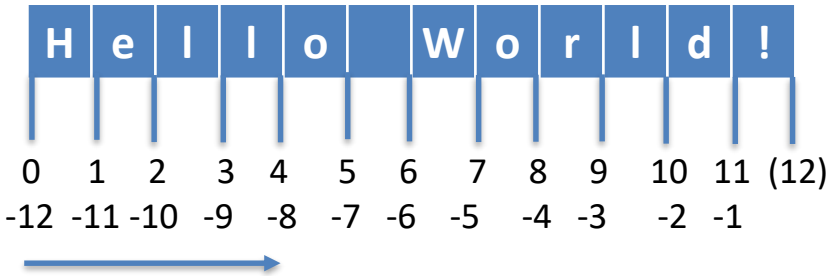
Manipulating a String (of characters) variable 2/2

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

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



- Applying operators directly to values is of very limited use.
  - Luckily, we can apply them to variables and assign the evaluation of the resulting expression to a new variable.

```
a_new_number = (a_number + another_number + 10) / 2
print(a_new_number)
```

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

```
greeting = "Hello"
place = "World!“
message = greeting+" "+place
print(message)
```

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

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

  - However, you cannot concatenate a String and another object unless you cast the latter as a String first using *str()*.

```
print("Hello it’s "+2020) #Check the error code.
```

Booleans

- An important simple data type we have not explored yet is boolean (*bool*).
  - Booleans have only two possible values: True or False*.
  - In Python, every data type inherently has a True or False value. 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…).

*In Python, True and False are capitalized (no quotes!)

Expressions and Operators revisited

- To compare two values or expressions, we can use comparison operators.
  - As with mathematical operators, many are fairly 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**…

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

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

```
number = int(input("Is this number negative or even"))
print(number<0 or number>100) #True if either 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.
```

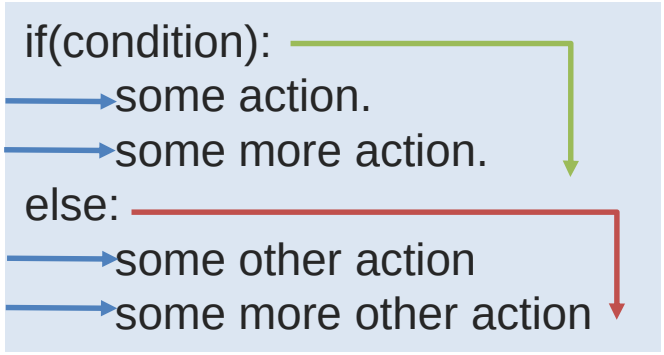
- Boolean still might feel of limited utility at this point. In fact we haven't achieved anything of much use up to this point. This is where **flow control** comes in, starting with **IF statements**.

- 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.

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

if number % 2 == 0 : #when True, the block of code below is executed.
        print(number, "is even")
else:                        #Otherwise this block of code is.
        print(number, "is not even")
        number += 1 #This is neat shorthand for number = number + 1
        print("well, now it is an even", number)
```

- Here is the general syntax:



- 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).

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

if number == 0 :
        print("is", number)
elif number > 0 :
        print(number, "is positive")
else:
        print(number, "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*.

*…Too many ELIFs might be a sign there's a better solution.

There can only be one True statement

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

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

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

  - However, the number 4 is even. Two statements therefore could evaluate to True. So what happens?
  - In Python, as in other programming languages, only the first True statement is evaluated.

- 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
my_name = "Matthieu"
print("yes, it's...")
for letter in my_name:
        print(letter)
        print("...")
print("…my last name as well?")
```

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.

Range() creates a sequence of numbers to iterate through.

The WHILE loop

- 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 += 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.
```

Part I challenge

- 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 go from there.

*It even happened to me.

Thanks for listening, see you Monday for Part II!

- 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**.

```
product = 1
for number in [2, 100, 0, 4.3, 6]: #This is a list, more on these in part II
    if(number==0):
        product = 0    #Notice the double-indentation.
        break          #Multiplying anything by 0 gives 0 again, exit loop
    else:
        product*= numbers
print(product)
```

  - It is also possible to simply skip to the next iteration using the keyword **continue**.