Introduction to

Matthieu Miossec, PhD (@RealMattJM)
Bioinformatics Core
https://www.well.ox.ac.uk/people/matthieu-miossec

28/03/2022
(Part II: 31/03/2022)

- 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 3.10.4 released last Thursday).

- To understand the driving philosophy behind Python, let's write our first line of code.

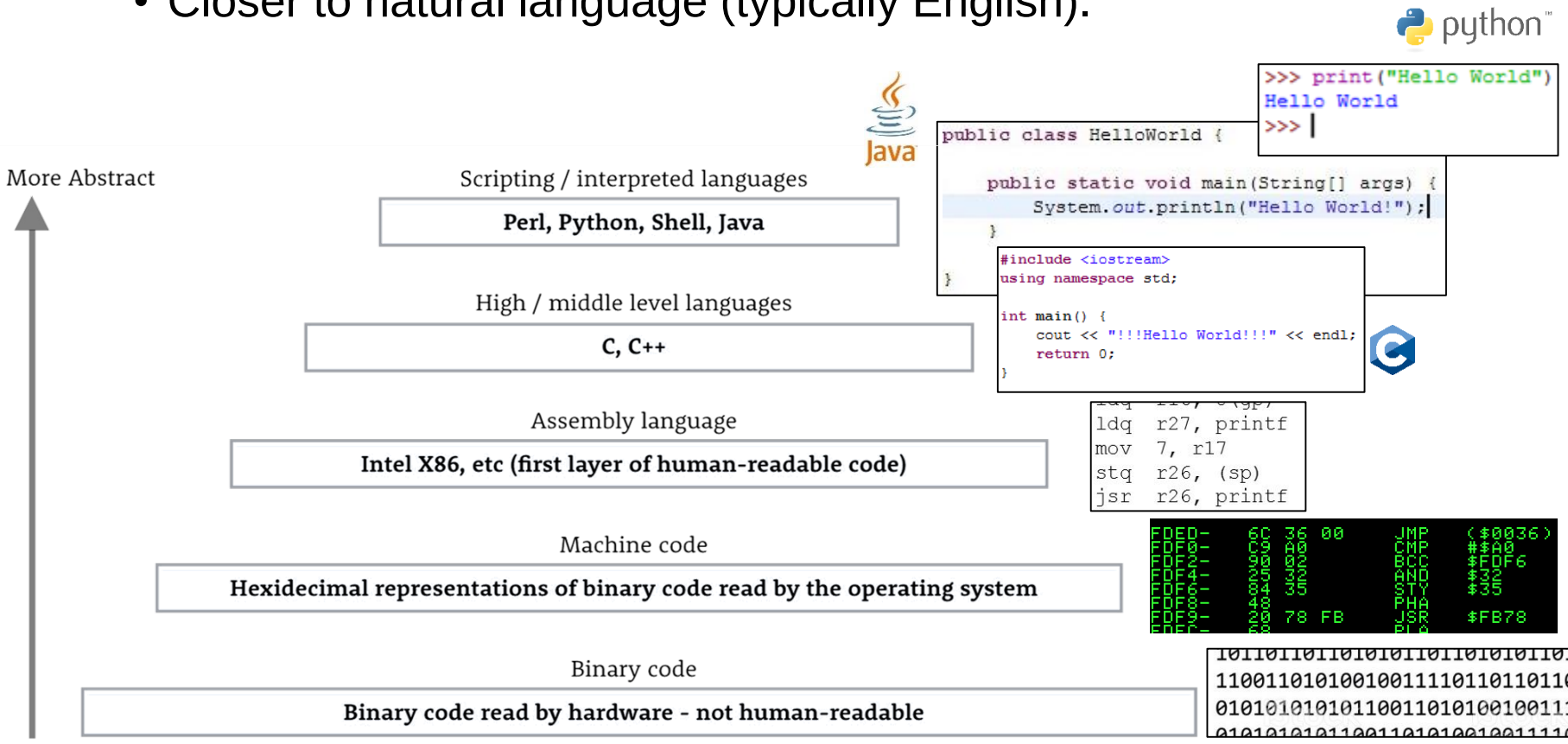
```
import this
```

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

*Please don't use Python 2, support 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?

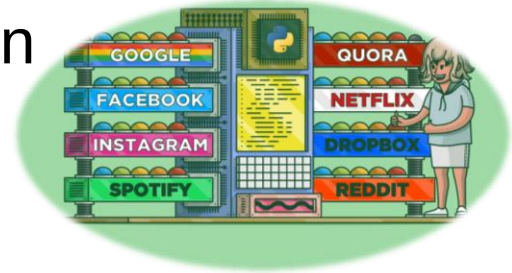
BIG DATA INSTITUTE

UNIVERSITY OF OXFORD

Mar 2022	Change	Programming Language	Ratings
1	▲	Python	14.26%
2	▼	C	13.06%
3	▼	Java	11.19%
4		C++	8.66%
5		C#	5.92%
6		Visual Basic	5.77%
7		JavaScript	2.09%
8		PHP	1.92%
9		Assembly language	1.90%
10		SQL	1.85%

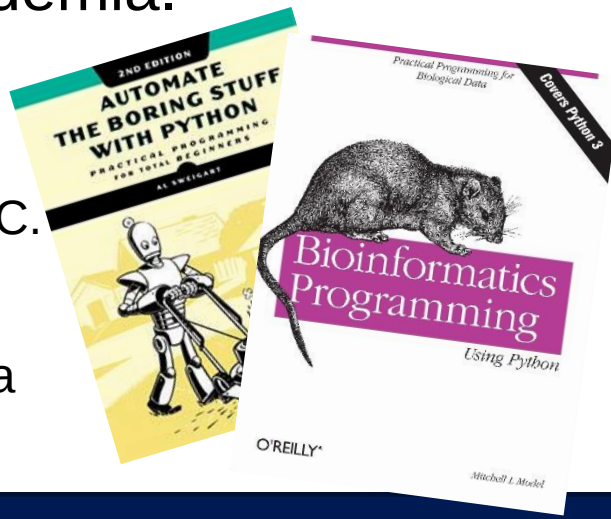
(source: TIOBE programming community index as of March 2022)

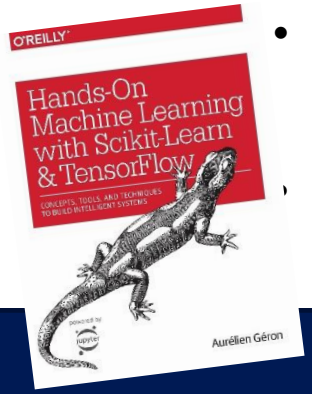
- One of the top programming languages right now. With applications in a variety of industries.



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

- Popular in academia.





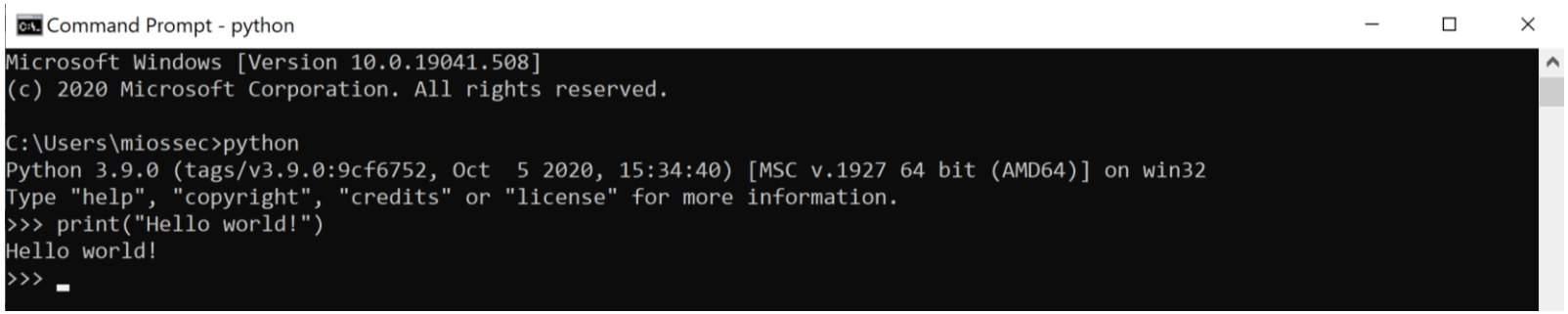
- Easier to start coding in Python than Java or C. Large community of users.
- A popular choice for scientific computing, data science and machine learning.

Ways to run Python –Interactive mode–

BIG DATA INSTITUTE

UNIVERSITY OF OXFORD

- With Python installed* on your machine, you can start writing and executing Python code directly into the command line terminal.
  - **Interactive mode**: In a terminal, type either *python* or *python3*. This will start the **Python prompt/shell** or **REPL** (Read-Evaluate-Print-Loop) in which you can type individual commands:

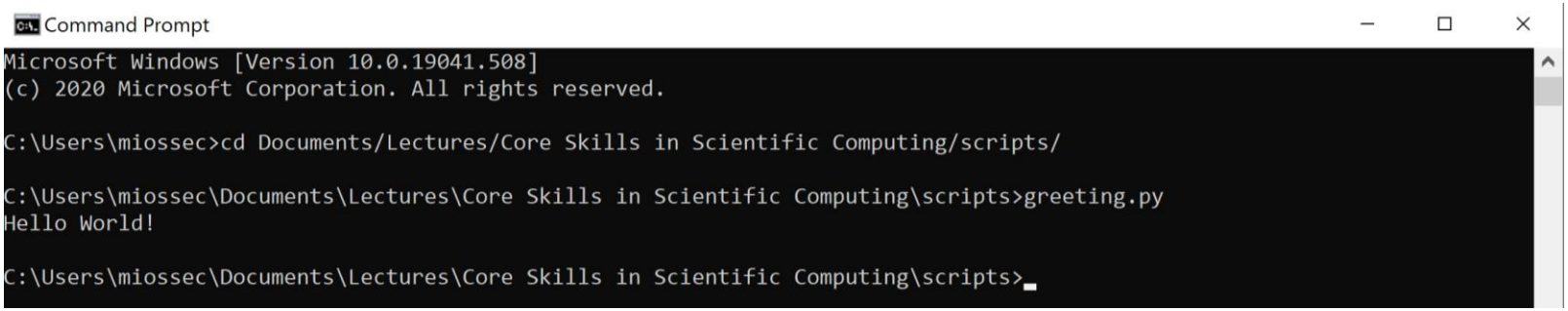


  - When you're done, type *quit()* to exit.

*Not required for this session, although I highly recommend you do this at some point.

Ways to run Python –script mode–

- A REPL is a good place to test single lines of code, but to execute and re-use 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 and execute your code within that environment!

Ways to run Python –IDE–

- An **integrated development environment** (IDE) brings together an interface with which to write code, a kernel to execute said code and a combination of useful tools to assist in various related tasks.
  - Recurring features include debugging, auto-complete, linting (checking for potential sources of errors as you write code), file-management and version control.
- We will be using the online IDE Replit. Beyond this workshop, a desktop environment is highly recommended.
  - This includes IDEs that are either compatible with (e.g. Atom, Sublime, Eclipse) or made for Python (e.g. Mu, PyCharm, Spyder, PyDev and JupyterLab). A good starting place: IDLE*, created **in** Python **for** Python.

*The extra L is for learning, but really it's a nod to Monty Python's Eric Idle.

Let’s start coding!

A challenge to start thinking about…

- By the end of today's session you should have all the necessary tools to implement the following task:
  - Write a script that takes the numbers 1 through 100, but 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.
- 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 data types, we will start by looking at *integers* (*int*).
  - We can use several familiar **operators** to join **values** into **expressions** which **evaluate** to a single new **value**:

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

```
7 * 3  # 21
7 / 3  # 2.3333333333333335
```

  - The above division expression evaluates to a *floating-point number* (*float*). If we want the fractional part of a division ignored, we can use //, or if we want the modulus/remainder of a division %.

```
7 // 3  # 2, integer division/floored quotient
```

```
7 % 3  # 1
```

  - We can also raise values to a given power using **.

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

- We can use the same operators to work with *floats* or a combination of *ints* 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, not 26
type(5.2 * 5)  # Type() shows <class 'float'>
```

  - If we want an value of type *int* (integer) again 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 from 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 double quotes is another common data type: *string* (i.e. string of characters, abbr. *str*) or *str*. We can also use single quotes to denote a *string*.

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

  - If the *string* itself contains an apostrophe, you can either escape the character (using \) 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 include several comma separates values into *print()* and get a single output message.

```
print(“What a",'wonderful',"World!") # Notice spaces between each element.
```

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

```
print("The year is", 2021+1)
```

  - …including *Booleans* (more on these soon!).

```
print(2+2<4) # This will evaluate to False
```

  - This is useful for creating a message that varies slightly between runs. Let’s create a **variable** during execution with the help of a new function, *input().*

```
adjective = input("Your input here:")
print(“What a", adjective, "World")
```

BIG DATA INSTITUTE

UNIVERSITY OF OXFORD

- A **variable** provides a way to store and access a value without needing to know where it is located in computer memory. An **assignment statement** looks like this:

```
word = "bird" # Takes the form variable = value
```

  - In most programming languages, we can assign the value of a variable to another variable via an assignment statement with both.

```
a_num = 5
b_num = 3
a_num = b_num
```

```
print(a_num)  # Outputs 3
```

  - **Only the value is assigned**. Future changes to one variable won't affect the other afterwards.

```
b_num = 7
print(a_num) # Still 3
```

- Using operators directly on values is of limited use. We can apply them to variables and assign the resulting value to a new variable.

```
c_num = (a_num + b_num + 10) / 2
print(c_num)
```

  - With *strings*, the + operator takes on new meaning: We use it to join (**concatenate**) *strings* together.

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

The * symbol (followed by an *int*) is used to repeat a *string* a given number of times.

```
print(greet*3) # HelloHelloHello
```

- You cannot concatenate a *string* and another data type unless you cast the latter as a *string* first using *str()*.

```
print("Hello it's "+2022) # What does the error code say?
```

Comparison Operators

- To compare a value or expression against another, we use comparison operators. Many will already be familiar.

```
a_num, b_num, word = 4, 10, "hello" # A quick Python trick…
print(a_num < b_num) # Less than. True
print(a_num > b_num) # Greater than. False
print(b_num >= a_num) # Greater or equal to. True
print(word <= "Hello") # Case matters. False
```

- The = operator is already reserved for variable assignment, to test equality we will always use == instead.

```
print(b_num == 10) #Equals to. True
```

- To find out if two expressions are not equal, we use !=

```
print(a_num != 10) #Not equal. True
```

- All these comparison operations lead to True or False values. What are these? Strings? Ints?...Booleans.

Booleans (and a break…)

- One final data type we'll look at today is *Boolean* (*bool*).
  - *Booleans* only have two possible values: True or False*.
  - Despite their simplicity, *Booleans* are very powerful with when used for **flow control** (i.e. IF statements and WHILE loops…).
- In Python, every data type inherently has a True or False value.
  - Little break 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…
# What about floats?

print(bool("Hello")) # True
# What could possibly cause False for a string?
```

*In Python, True and False are capitalized (and no quotes or it's a *string*)

Logical statements

- Comparison expressions/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 are true. False otherwise.
```

- …**or**…(this is an **inclusive or**, 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.
```

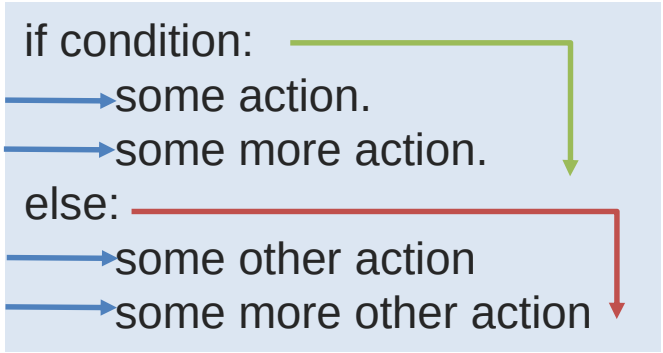
- We can now introduce **flow control** comes in, starting with **IF statements** and begin to write flexible code!

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

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

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 odd")
```

- Here is the general syntax:



- Notice the indentation. This is how you separate **blocks of code** in Python. Only what is indented after IF (its clause) will be executed if the condition is true. Otherwise, only the block after else (another clause) will be executed.

- What if we want to evaluate more than one conditional statement? For this we use **ELIF** (contraction of else and 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.
  - A flow control structure 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

- More than one condition could be true. Say we want to know if a number is 4, or otherwise 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, only the first True statement is evaluated.

The FOR loop

- 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*.

```
# How do you spell Python user?
you = “Pythonista"
print("it's...")
for letter in you:
        print(letter)
        print("...") # Pause
```

```
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 = 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 build on that

*It happened to me once.

Don’t worry if your solution isn’t optimal…

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

- You can always revisit your code and make incremental improvements. This is the way.

How are we doing for time?

Manipulating a string (of characters) variable 1/2

- A *string* 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(greet)) #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*.

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

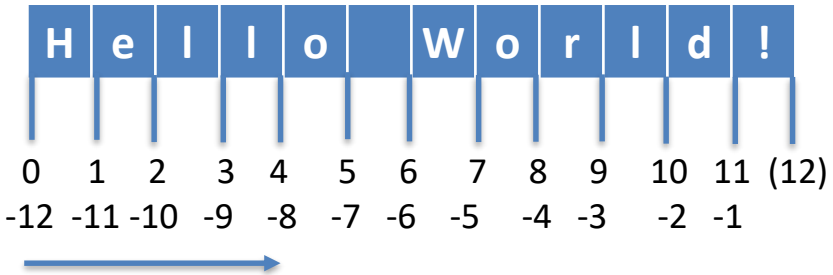
*This is the source of many errors, so try to keep that in mind.

- 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.
```



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…
       """)
```

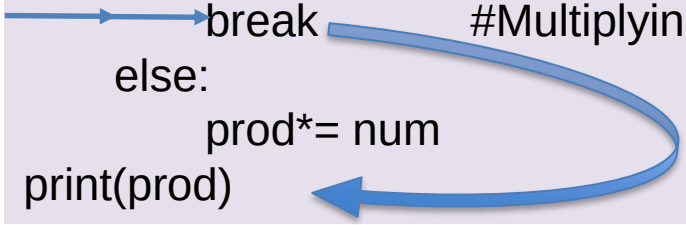
  - For everyday programming tasks, single or double quotes are usually adequate. We can always skip to a new line using *\n*.

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

Extra: Breaking out of a FOR loop early

- 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?

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