Programming with

Dr Matthieu Miossec
Senior Bioinformatician @ Wellcome Centre for Human Genetics
https://www.well.ox.ac.uk/people/matthieu-miossec

30/10/2023

Part II: 06/11/2023

- Python is a **high-level programming language** originally developed by Guido Van Rossum in 1991.
  - The last major language revision dates back to 2008 and led to **Python 3***. The language is continually evolving (latest version: 3.12.0, released a month ago).

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

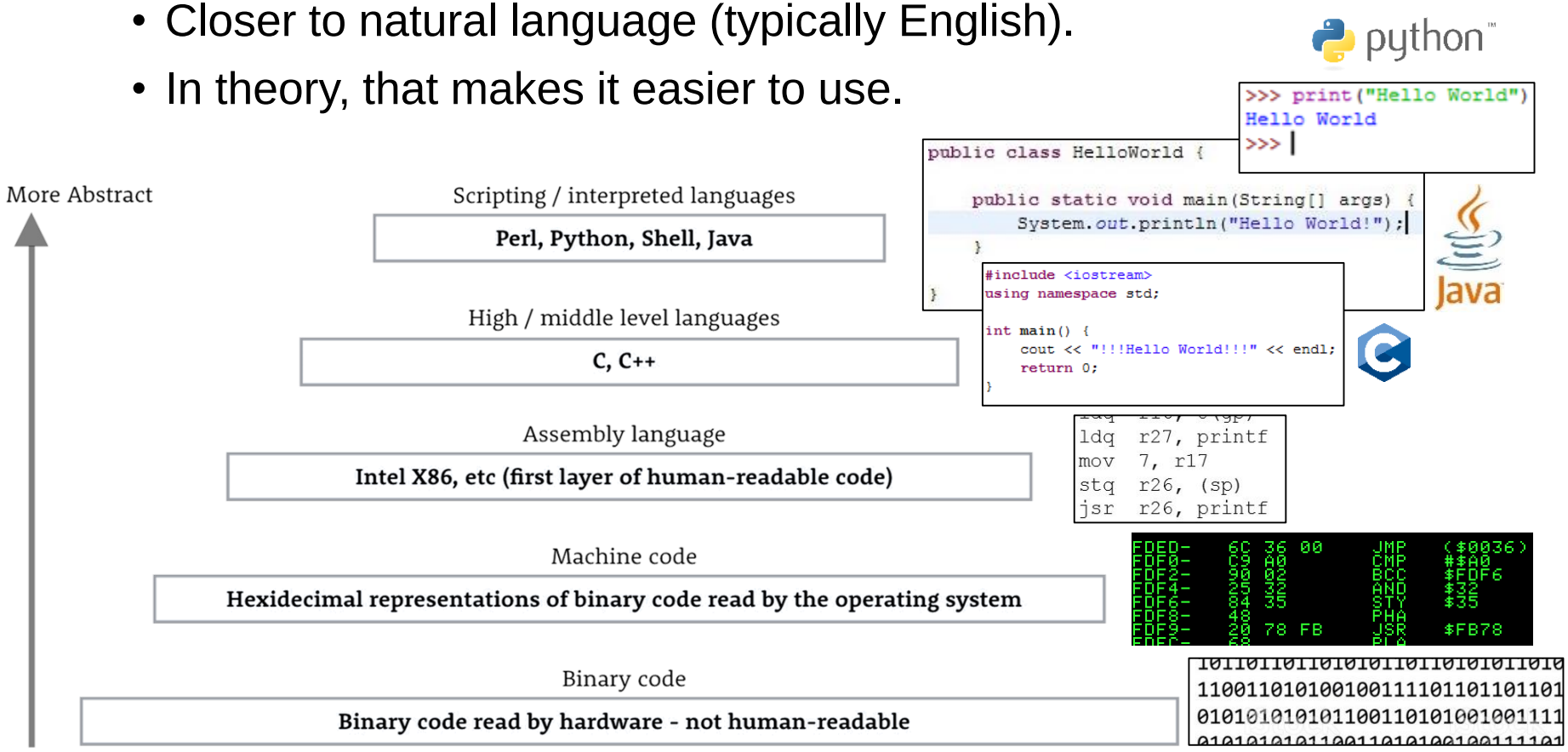
```
import this
```

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

*Please don't use Python 2, support has been discontinued. Don't worry, there will be no Python 4

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).
  - In theory, that makes it easier to use.



Why learn Python?

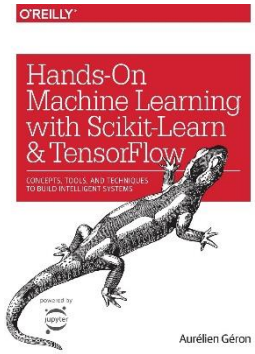
Oct 2023	Programming Language	Ratings
1	Python	14.82%
2	C	12.08%
3	C++	10.67%
4	Java	8.92%
5	C#	7.71%
6	JavaScript	2.91%
7	Visual Basic	2.13%
8	PHP	1.90%
9	SQL	1.78%
10	Assembly language	1.64%

(source: TIOBE programming community index, Oct 2022)

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

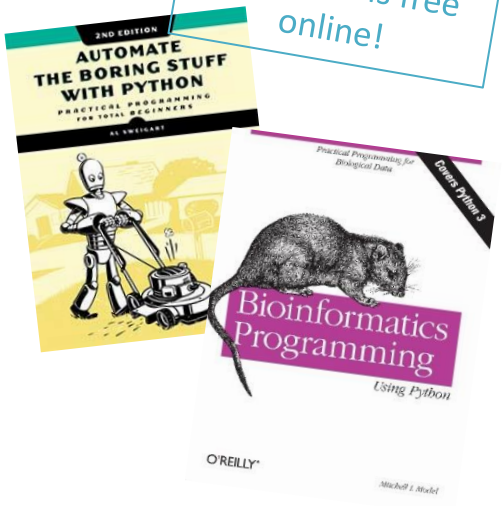
(see: https://realpython.com/world-class-companies-using-python/ for examples)

- Popular in academia
  (e.g. from our group: GeneFEAST a tool for gene functional enrichment analysis was written in Python)



- Easier to pick up than languages like Java or C.
  - Clear, simple, straightforward syntax.
- Large community of users (extensive libraries).
  - Enables diverse applications.
- A popular choice for scientific computing, data science and machine learning.

This book is free online!



- By the end of today's session you will have all the necessary tools to implement the following task:
  - Write a script that runs through the numbers 1 to 100, prints *fizz* (to screen) for multiples of 3, *buzz* for multiples of 5 and *fizzbuzz* for multiples of 3 and 5 (…and prints numbers otherwise).
- As we cover various aspects of Python today, think about what might be useful in addressing the following:
  - Identifying multiples of 3 or 5 (or both).
  - Running through a list of numbers.
  - Performing an action when a specific condition is fulfilled.

Let’s start coding!

- Python has several data types. Let's start with *integers* (*int*).
  - We 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 new values above are themselves integers except for the last example (division) which 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 *float*s, as well as a combination of *int*s 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 to convert a value to *int* again we use **casting** which converts a data type into another. Here with the function *int()*:

```
int(5.2 * 5) # 26
```

  - 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
```

type(), int() and round() are built-in Python functions. More on functions later.

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

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

  - The statement contained in double/single quotes is another common data type: *string* (i.e. string of characters) or *str*.
  - If the *string* you want to write contains an apostrophe, you can either escape the character (using \).

```
print('Season\'s greetings')
```

  - or use double quotes.

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

  - or vice versa.

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

```
print('1')
```
These are *strings* rather than *ints*
```
print('2+2')
```

- In Python, we can include several comma separated values into *print()* and get a single output to screen. These values can originate from a mix of data types and expressions.

```
print("The year is", 2022+1, "AD") # notice the added space between each value
```

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

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

  - This is useful for creating an output message that varies between executions. Let's give a **variable** a value determined during execution with the help of a new function, *input().*

```
adjective = input("Your input here:") # saves input to adjective
print("What a", adjective, "World") # used here
```

- A **variable** is a way to store and retrieve a value without explicitly referring to its memory address within a computer. 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.

```
x = 5
y = 3
x = y
```

```
print(x)  # outputs 3
```

Python Pros can check out the infamous Walrus Operator :=
https://youtu.be/KN2TTiGpDvM

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

```
y = 7
print(x) # still 3
```

Expressions with variables

- We can combine variables (and values) into expressions using operators as before. The resulting value is itself typically assigned to a variable.

```
avg = (x + y + 10) / 3
print(avg)
```

  - With *strings*, the **+** operator takes on new meaning: it joins (aka. **concatenates**) *strings* together.

```
greet = "Hello"
place = “world!"
msg = greet+"\t"+place
print(msg)
```

The **n* operator (*n* a given number) is used to repeat a *string n* times.

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

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

```
print("Hello it’s "+2022)
```

What does the error message say?

\t stands for tab. If you want to skip a line there’s also \n for newline

- Let's create three variables in one line with a quick little Python trick.

```
x, y, word = 4, 10, "hello"
```

- To compare values or expressions, we use comparison operators, many already familiar (but let's first create 3 variables with one line of code).

```
print(x < y) # less than. True
print(x >= y) # greater or equal to. False
print(word <= "Hello") # works with Strings. Case matters. False
```

  - The = operator is reserved for variable assignment, to evaluate equality we use == instead.

```
print(y == 10) # equals to. True
```

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

```
print(y != 10) # not equal. True
```

  - All these comparison operations lead to **True** or **False** values. What type are we dealing with? *String*? *Int*?...*Boolean*.

Booleans (and a quick break?)

- One final data type to look at today is *boolean* (*bool*).
  - *Booleans* only have two possible values: True or False*.
  - Despite their simplicity, *booleans* are very powerful when used for **flow control** (i.e. IF statements and WHILE loops…).
- In Python, every type inherently has a True or False value, a 'truthiness'.
  - Little break challenge: Can you figure out what is **True** and what is **False** for *int* and *string* using *bool()*? (yes this is casting)

```
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**… (**inclusive or**, i.e. 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_val = False
print(not some_val)  # True if False. False if True.
```

- We can use the 'truthiness' associated with types to write concise code.
(though of course you don't have to. No pressure!)

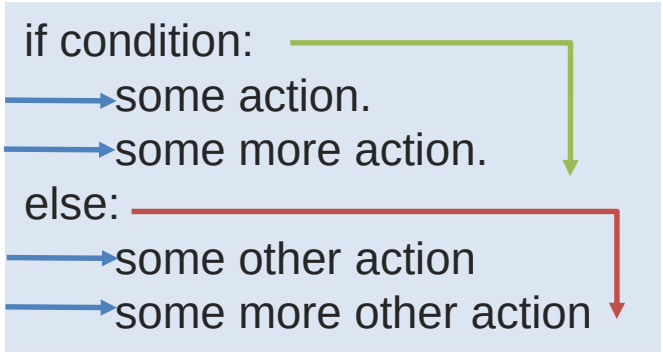
```
num = int(input("Is this number odd?"))
print(bool(num%2))  # can you figure out how this works?
```

- So far, we've written a lot of statements that can fit into one or two lines. However, an **IF statement** takes up several 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 an even number")
else:                       # otherwise this block of code is.
        print(num, "is an odd number")
```

- The general syntax:



- Notice the indentation. This is how you separate **blocks of code** in Python. Only what is indented after IF, i.e. its **clause**, will be executed if the condition is true. Otherwise, only the block indented below 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(“Done.")
```

  - Notice this final print statement appears to be aligned with the entire IF structure. This is because 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.

- Using ELIF, we run into the problem of more than one condition being true, e.g. Is a number 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, only the first True statement is evaluated.

- One of the big benefits of programming is that it allows us to perform repetitive actions in an automated fashion.
  - One way to apply an action repeatedly 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*.

```
lyric = “YMCA"
for i in range(2):
    print("it's fun to be in the")
    for letter in lyric:
        print(letter*5)
        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.

- 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 # equivalent to i = i+1
print("done!")
```

The variable that is being continually evaluated needs to be initialised before the loop and changed within the loop.

  - When writing WHILE loops, be very careful or else your loop could be infinite! (here i remains 0 until program crashes, not ideal)

```
# 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
```

You're ready for Part I challenge!

- A common interview question* that uses some of the things we've covered today:
  - Write a script that runs through the numbers 1 to 100, prints *fizz* (to screen) for multiples of 3, *buzz* for multiples of 5 and *fizzbuzz* for multiples of 3 and 5 (…and prints numbers otherwise).
- 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.

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

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

How are we doing for time?

Slicing 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!

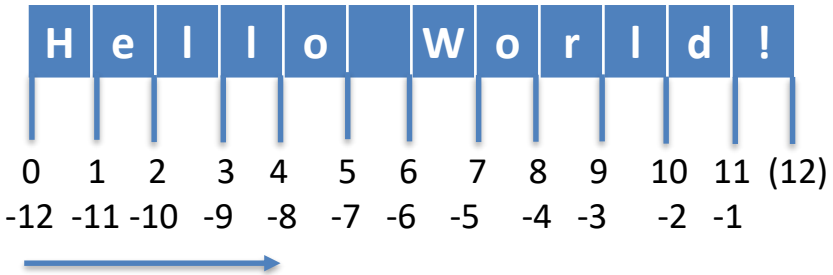
*This is the source of many errors, so try to keep that in mind.

- 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[:])
```

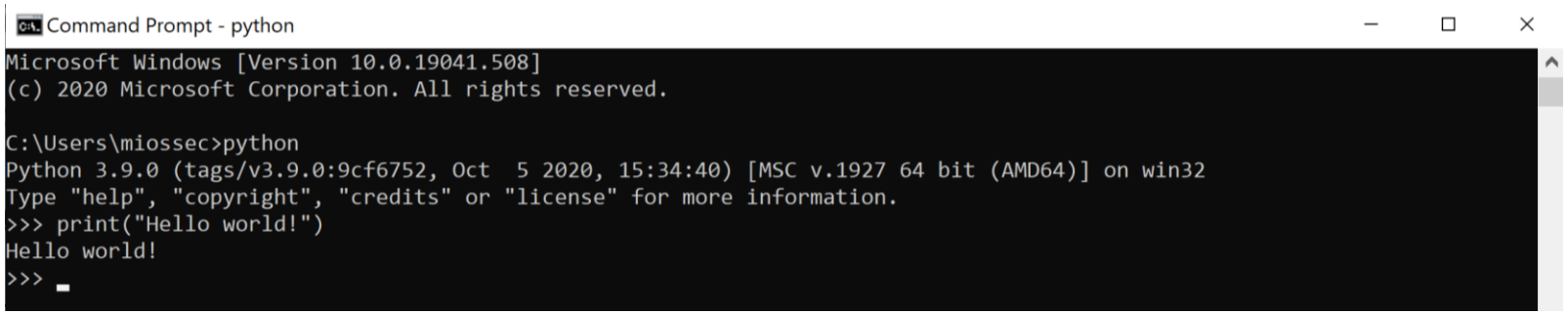
  - Although this might not seem useful for now, it is also possible to use negative indices which refer to elements from the end of the *string*.

```
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 in a bit for

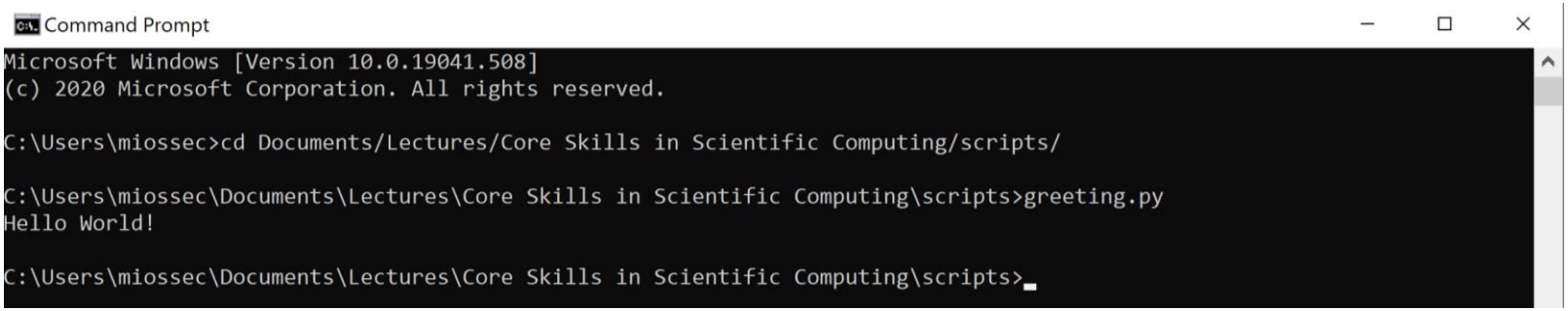
- With Python installed on your machine, you can start writing and executing Python code directly into the command line terminal.
  - **Interactive mode**: In a terminal, type either *python* or *python3*. This will start the **Python prompt/shell** or **REPL** (Read-Evaluate-Print-Loop) in which you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



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

Extra: 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!

- 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.
- For an online IDE, use Replit. Beyond this course, a desktop environment is highly recommended.
  - This includes IDEs 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.

- What if we want to include both types of quotes in a *string*?
  - Use triple quotes! (single/double x3)

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

  - Triple quotes let's you write and print over lines the way it appears on your screen. This is a feature unique to Python (last I checked…).

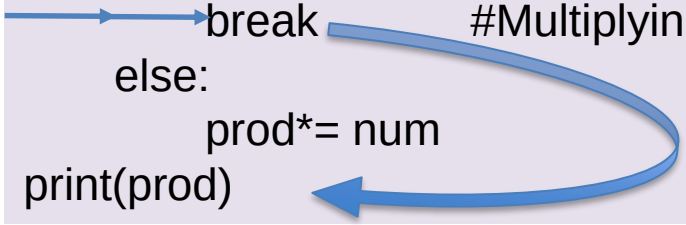
```
print('''Here is some DNA:
        …A T G A A C…
           | | |   | | |
        …T A C T T G…
        ''')
```

  - For everyday programming tasks, single/double quotes are adequate. We can always go 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? It’s because of snakes isn’t‘ it

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