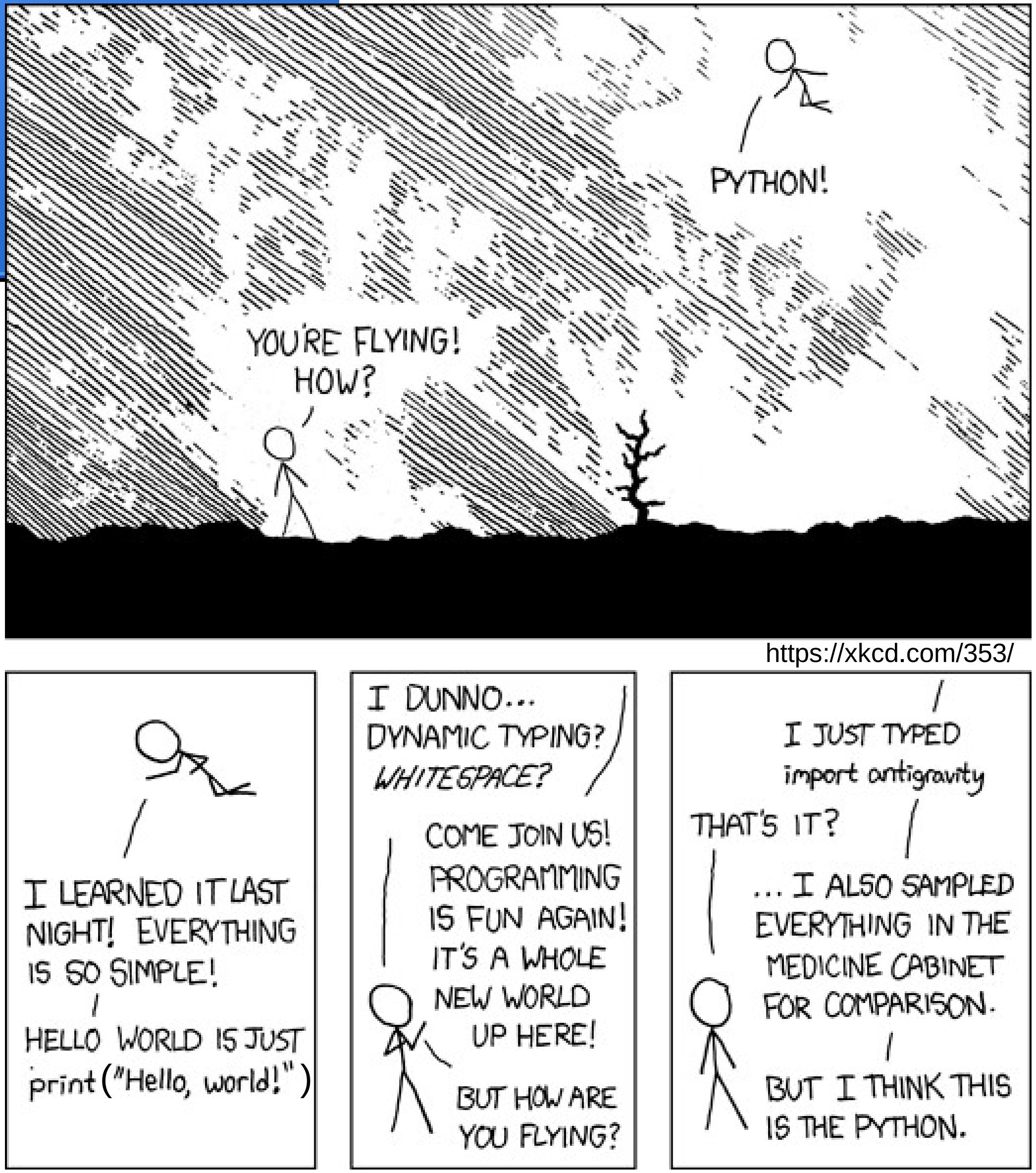
Introduction to Python


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print("Hello, world!")
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

https://xkcd.com/353/

EIEIOO
Tuesday May 4th, 2021
Jean-François Caron

About Jean-François

- Post-doctoral researcher with NEWS-G at Queen’s
- Working on making a neutron beam at our small accelerator for calibrations (see WNPPC 2021 talk)
- Previously worked on Mu2e tracker module construction at U.Minnesota, Fermilab, and Houston
- Ph.D. on the SuperB drift chamber (R.I.P.)
- I do gas ionization particle detectors, instrumentation, electronics, hardware, etc.

- Bicycle repair, bicycle touring, long-distance rides
- Downhill and cross-country skiing
- Roguelike computer games, science fiction books
- Cooking and home fermentation

What is Python?

- Python defines a programming language.
  - Interpreted language, meaning it does not get compiled to native machine instructions
  - Multi-paradigm, meaning you can implement almost any kind of programming style (*e.g.* functional, object-oriented, imperative)
- CPython is the official *implementation* of the Python language written in C.
  - Since it’s free & open-source, there are alternative implementations: PyPy, Jython, MicroPython...
  - Comes built-in as a system tool in Linux and macOS.

More About Python

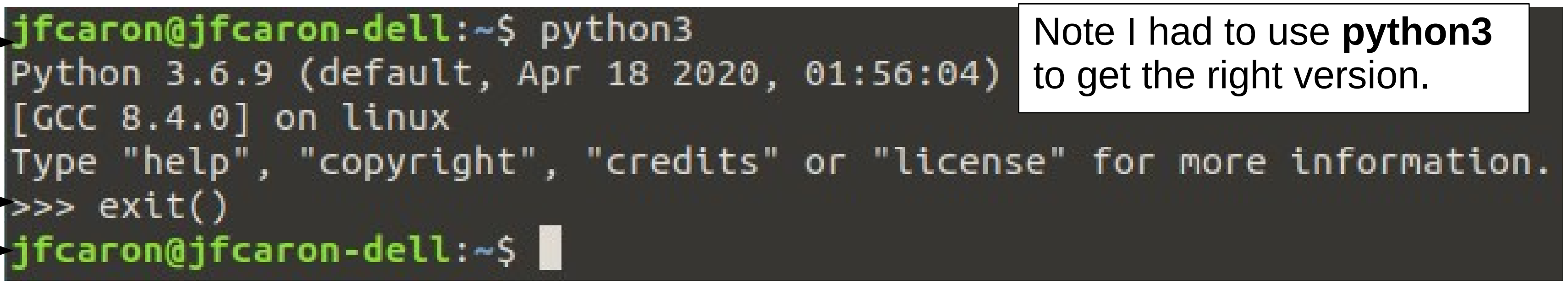
- It is easy to add non-Python extensions to Python. *E.g.* numpy is a numerical matrix library that is written in FORTRAN.
- This makes Python a great “glue language” to combine different bits of software.
- It also lets you use systems written in a more-difficult-to-learn high-efficiency language (*e.g.* C++, FORTRAN) in a friendlier setting.
- Python itself is a capable language, but large applications in pure python can be slow.

Python Versions

1) Use python 3.x
2) Use python 3.x
3) The python that comes with current Ubuntu is 3.x. macOS currently has 2.x ***and*** 3.x.
4) The two are mildly incompatible.
5) You may have to run **python3** instead of just **python** at the command line.
6) When in doubt, use **python -V** and **python3-V** to test the version.

Python Interpreter

- When you run **python** in bash, you enter a *python interpreter*. A.k.a. python prompt, python shell.
- This is very similar to the bash shell, but it's oriented towards programming, whereas bash is oriented around files and running other programs.
- Type **`exit()`** and press enter, or Ctrl+D to return to bash.



Bash

Python prompt

Back to bash

```
jfcaron@jfcaron-dell:~$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
jfcaron@jfcaron-dell:~$ 
```

Note I had to use **python3** to get the right version.

Tip: you can type the partial name of a command or file then press **tab** and python will try to complete the name.
You can also press the up-arrow to repeat earlier commands.

Python as an Engine

You can write python programs in a text file, conventionally with .py extension. Then

python my_program.py

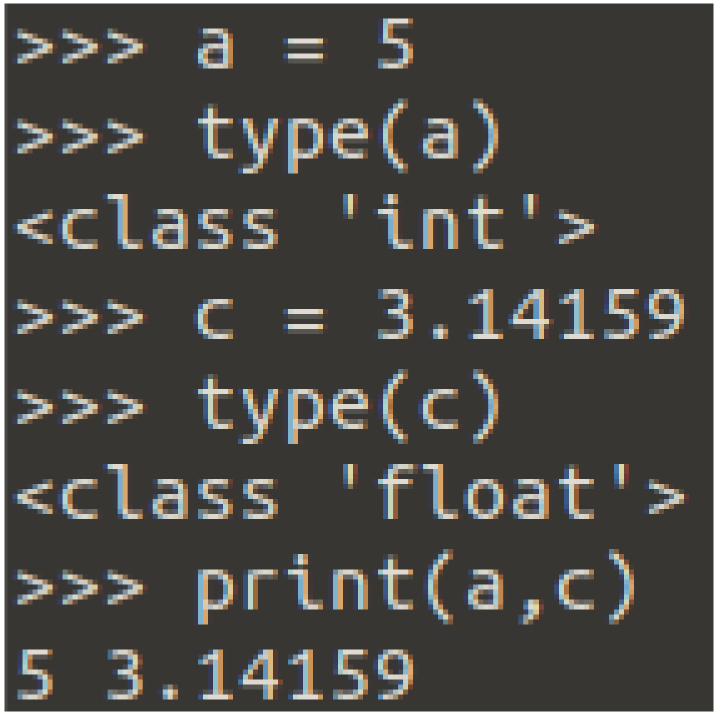
will run the program and exit.

If you meant to display something on a window, like a graph, then it will disappear when python exits!

Adding **-i** to the command will run the program then leave you at the interpreter.

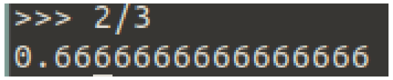
A good workflow is to use **-i** to experiment in the interpreter before adding new code into your program in a text file.

Basic Math



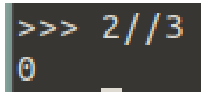
```
>>> a = 5
>>> type(a)
<class 'int'>
>>> c = 3.14159
>>> type(c)
<class 'float'>
>>> print(a,c)
5 3.14159
```

Python numbers behave as you might expect from a mathematical perspective.
It is less “computer-y” than in other languages like C.



```
>>> 2/3
0.6666666666666666
```

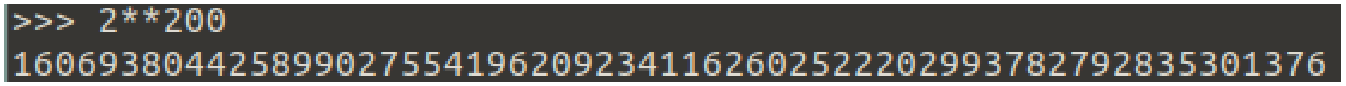
Dividing two integers does not truncate the decimal places, it returns a float.



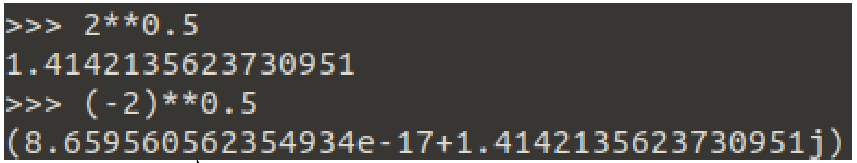
```
>>> 2//3
0
```

You can still do “integer division” with **//**.

Python integers are “BigNum”, meaning there is no maximum/rollover.
Python integers will just use more memory as they get really big.



```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376
```



```
>>> 2**0.5
1.4142135623730951
>>> (-2)**0.5
(8.659560562354934e-17+1.4142135623730951j)
```

real part + jmaginary part

You can even do square roots with non-integer exponents and get complex numbers.

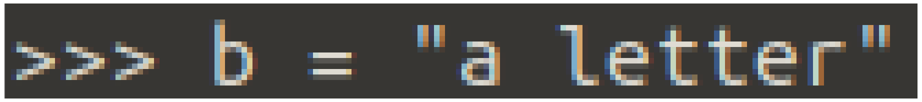
Floating point numbers are valid IEEE 754 floats, like every other language out there, with all their quirks.

*** / + -** do what you would expect.
// does integer division, **%** does modulo
****** does exponentiation (not **^**)
+=, -=, *= etc are “augmented assignment”:
a += b is the same as **a = a+b**.

Python Typing

- *Dynamic typing* means you don’t have to declare variables ahead of time, and a python *name* isn’t restricted to one type.
- *Strong typing* means you have to explicitly transform variables into different types.
  e.g. `2+’3’` won’t work (but `2*’3’` works...)
- Everything is an object in python.
- *Duck typing* means you can legally pass any object to any function.
  (If it quacks like a duck, it’s a duck)

Python Strings

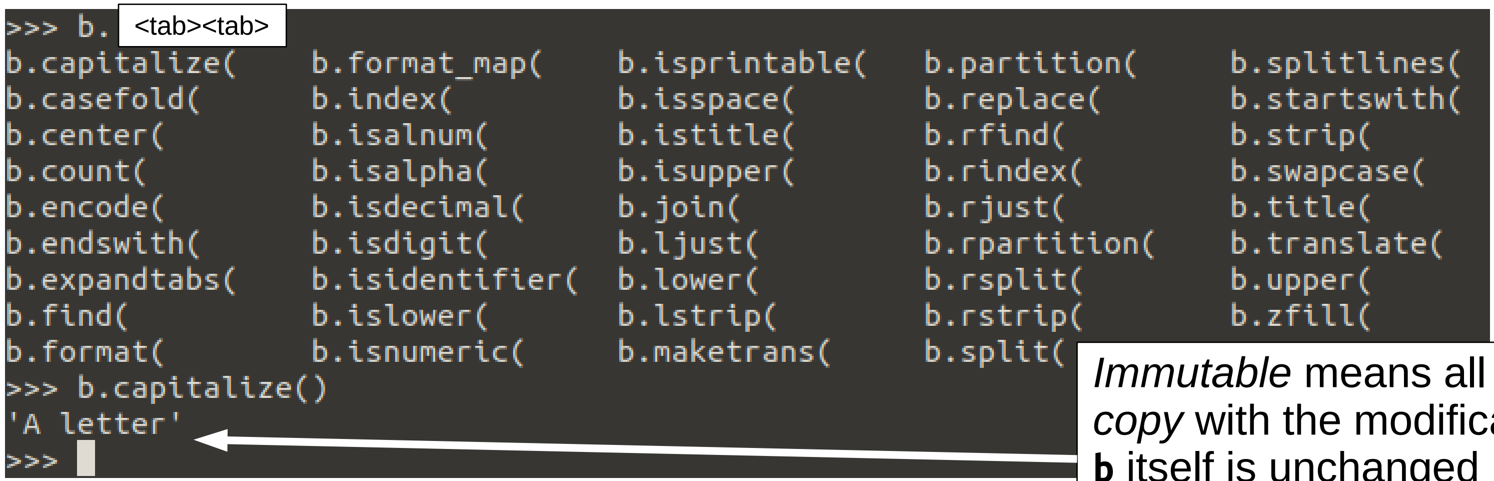


```
>>> b = "a letter"
```

You can make strings with ‘single quotes’, “double quotes”, and even ‘’’triple’’’ and “””sextuple””” quotes, but these are all equivalent (almost).

There is no bare character type in python, a single character is just a string of length 1.

Python strings are built-in and have lots of convenient “methods”.

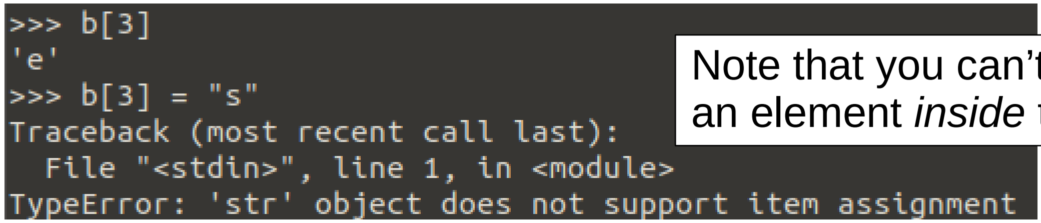


```
>>> b. <tab><tab>
b.capitalize(    b.format_map(    b.isprintable(   b.partition(     b.splitlines(
b.casefold(      b.index(         b.isspace(       b.replace(       b.startswith(
b.center(        b.isalnum(       b.istitle(       b.rfind(         b.strip(
b.count(         b.isalpha(       b.isupper(       b.rindex(        b.swapcase(
b.encode(        b.isdecimal(     b.join(          b.rjust(         b.title(
b.endswith(      b.isdigit(       b.ljust(         b.rpartition(    b.translate(
b.expandtabs(    b.isidentifier(  b.lower(         b.rsplit(        b.upper(
b.find(          b.islower(       b.lstrip(        b.rstrip(        b.zfill(
b.format(        b.isnumeric(     b.maketrans(     b.split(
>>> b.capitalize()
'A letter'
>>> 
```

Strings are:
- ordered
- homogenous
- immutable

Immutable means all string operations return a *copy* with the modification. **b** itself is unchanged here, we just got a new string with the capitalization.

Ordered means you can “index” into a string with square brackets and an integer. More indexing syntax is shown later.



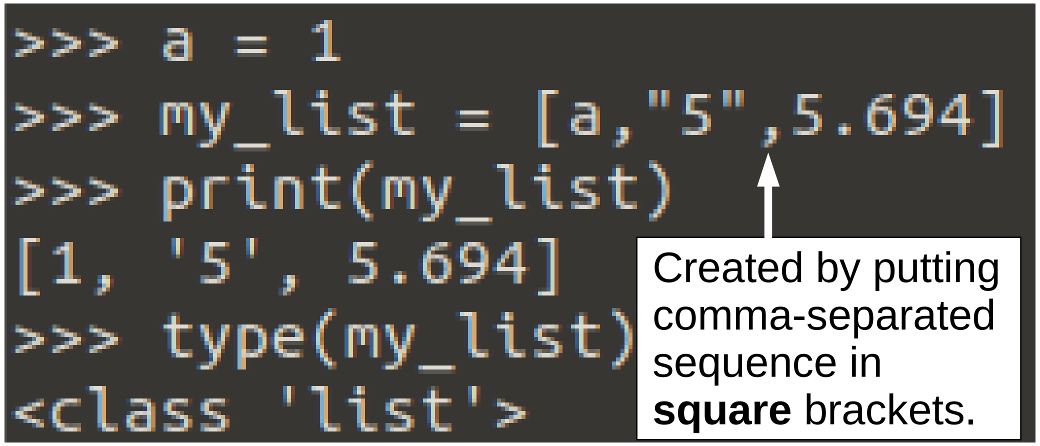
```
>>> b[3]
'e'
>>> b[3] = "s"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Note that you can’t change an element *inside* the string.

Exercise

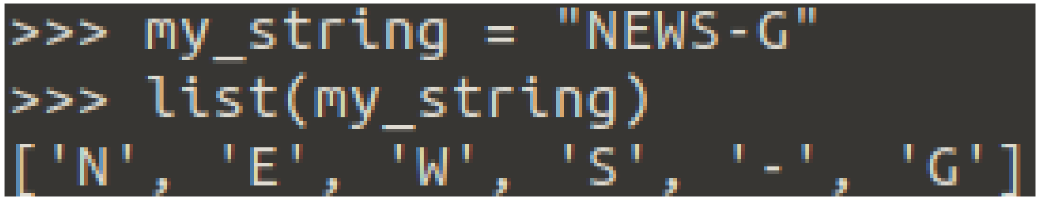
1) Write your full name as a string.
2) Get the length of this string.
3) Use the string’s **`.split`** method.
4) What is returned?
5) Type **`help(stringname.split)`**
    - These are like manpages for python.
    - The **`help`** command is great for functions (and modules...)
    - Not so great for other objects: *e.g.* try **`help(str)`**.

Basic Containers: Lists



```
>>> a = 1
>>> my_list = [a,"5",5.694]
>>> print(my_list)
[1, '5', 5.694]
>>> type(my_list)
<class 'list'>
```

Created by putting comma-separated sequence in **square** brackets.

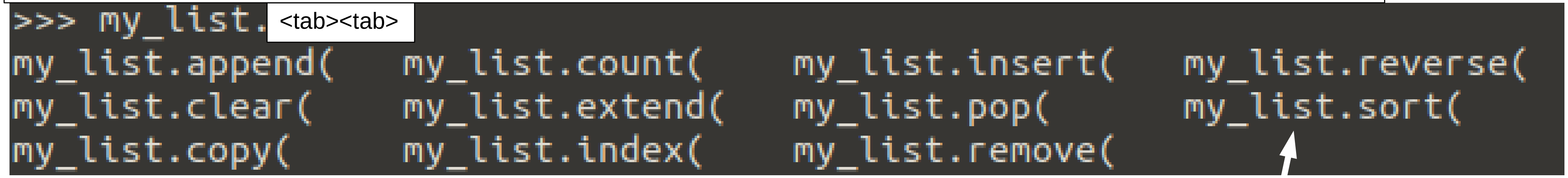


```
>>> my_string = "NEWS-G"
>>> list(my_string)
['N', 'E', 'W', 'S', '-', 'G']
```

You can also put any sequence into the **`list()`** function.

Lists are:
- ordered
- heterogenous
- mutable
- the most common container in python.

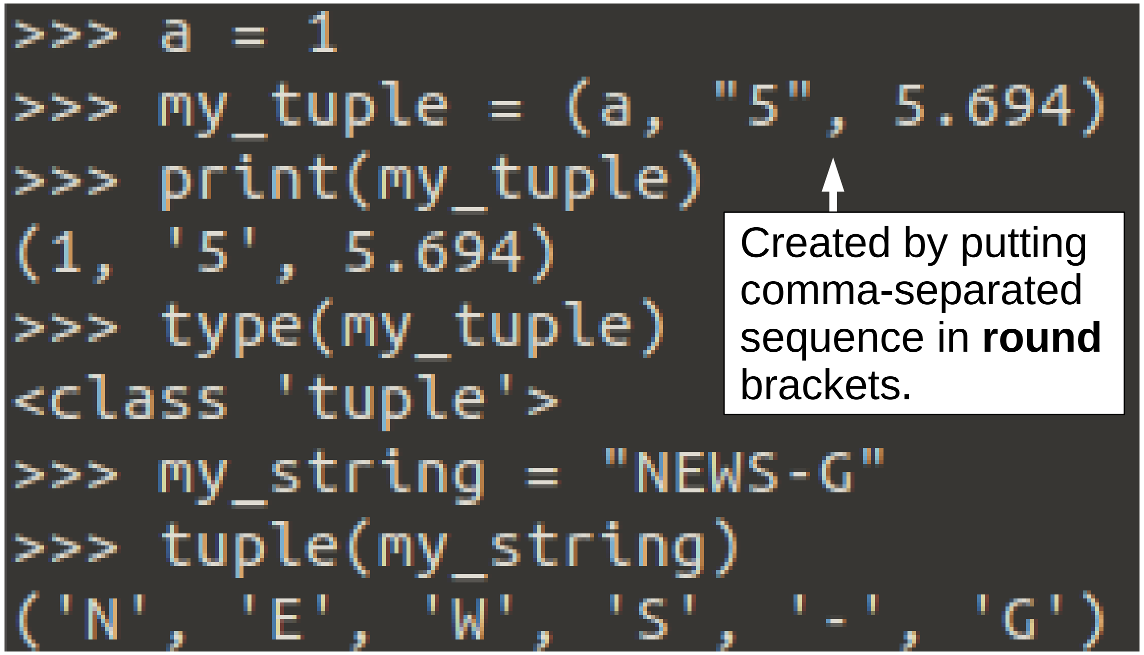
Lists, like most python objects, have a bunch of convenient and obvious built-in methods.



```
>>> my_list. <tab><tab>
my_list.append(    my_list.count(     my_list.insert(    my_list.reverse(
my_list.clear(     my_list.extend(    my_list.pop(       my_list.sort(
my_list.copy(      my_list.index(     my_list.remove(
```

Since lists are mutable, these modify the list itself.

Basic Containers: Tuples



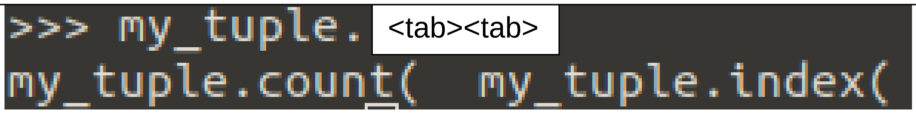
```
>>> a = 1
>>> my_tuple = (a, "5", 5.694)
>>> print(my_tuple)
(1, '5', 5.694)
>>> type(my_tuple)
<class 'tuple'>
>>> my_string = "NEWS-G"
>>> tuple(my_string)
('N', 'E', 'W', 'S', '-', 'G')
```

Created by putting comma-separated sequence in **round** brackets.

Tuples are:
- ordered
- heterogenous
- immutable
- the default container in python.

You can also put a sequence into the **`tuple()`** function.

Tuples are simpler than lists, they only have two methods.

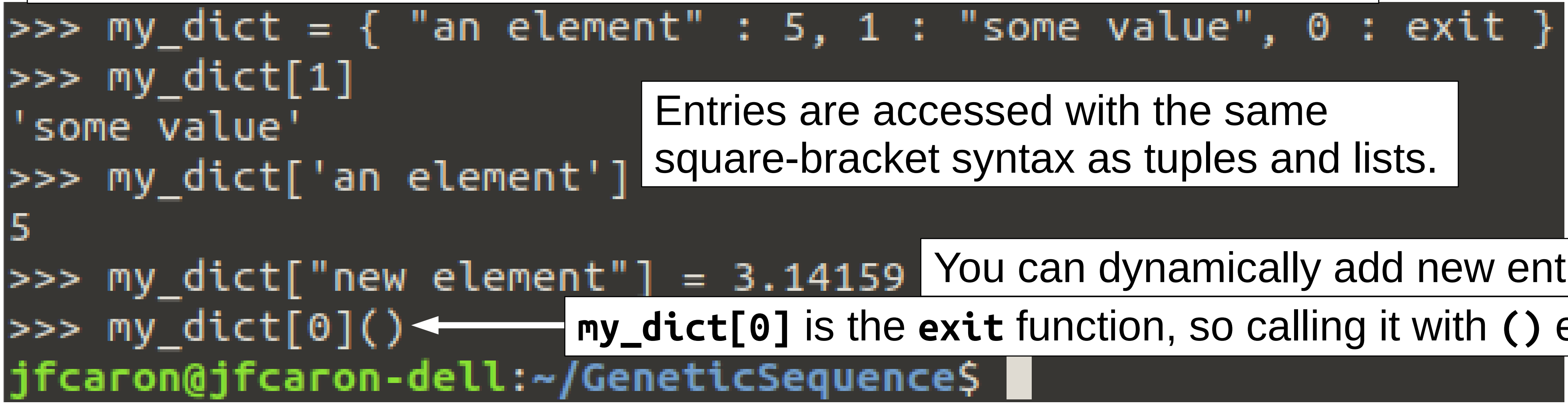


```
>>> my_tuple.
my_tuple.count(   my_tuple.index(
```

<tab><tab>

Basic Containers: Dictionaries

Dicts are created with **curly** brackets and take *key:value* pairs.



```
>>> my_dict = { "an element" : 5, 1 : "some value", 0 : exit }
>>> my_dict[1]
'some value'
>>> my_dict['an element']
5
>>> my_dict["new element"] = 3.14159
>>> my_dict[0]()
jfcaron@jfcaron-dell:~/GeneticSequence$
```

Entries are accessed with the same square-bracket syntax as tuples and lists.

You can dynamically add new entries.

`my_dict[0]` is the **`exit`** function, so calling it with **`()`** exits.

Keys must be *immutable.* You can use strings and tuples as keys, but NOT lists!

Values can be any object.

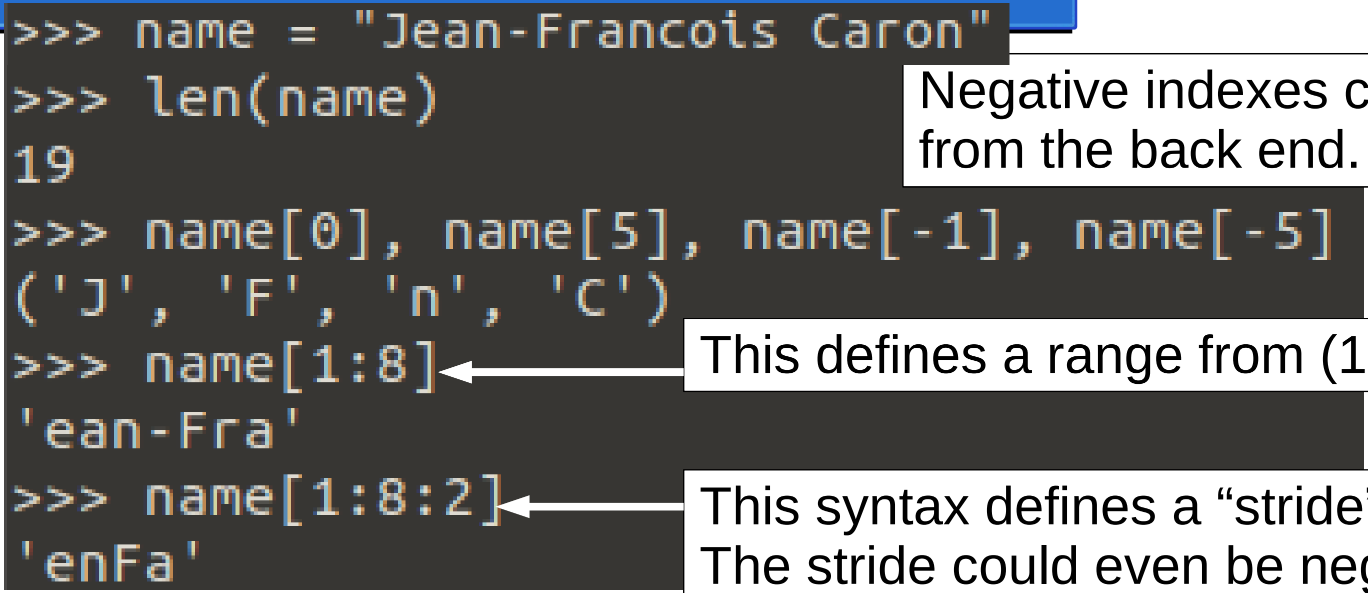
A lot of python internal mechanics are implemented using dicts *e.g.* **`globals()`**

Dicts are:
- unordered
- heterogenous
- mutable
- underutilized!

Exercise

- Find the length of the string with your full name. (use **`len`**)
- What are elements [0], [5], [-1], and [-5] of your name?
- What do you get when you ask for element [1:8]? What about [1:8:2]?
- Take the result of using **`.split()`** on the string with your full name.
- Create a dict with keys “first” and “last” (and other parts…) and put the parts of your name in it.
- Print the dictionary.

Exercise (2)



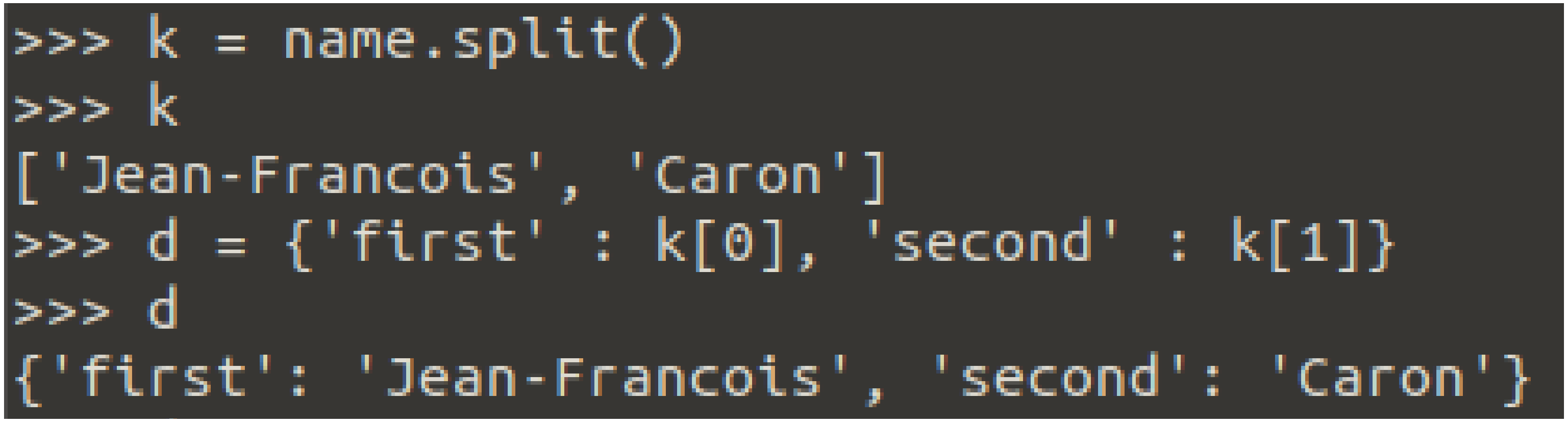
```
>>> name = "Jean-Francois Caron"
>>> len(name)
19
>>> name[0], name[5], name[-1], name[-5]
('J', 'F', 'n', 'C')
>>> name[1:8]
'ean-Fra'
>>> name[1:8:2]
'enFa'
```

Indexes start from 0.

Negative indexes count from the back end.

This defines a range from (1,8].

This syntax defines a “stride” of 2. The stride could even be negative to reverse a sequence.

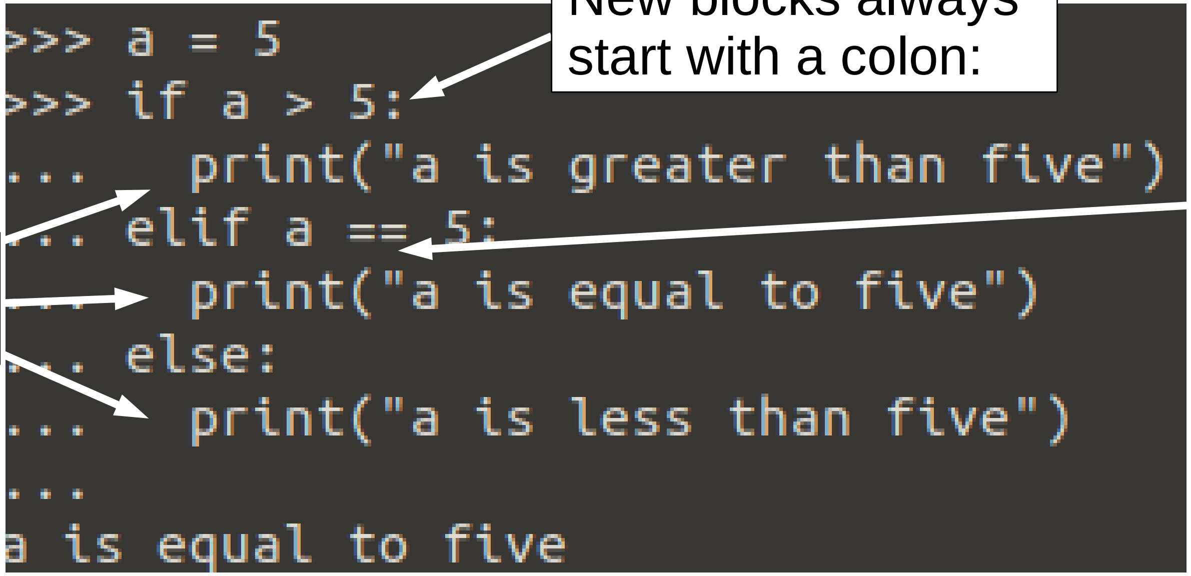


```
>>> k = name.split()
>>> k
['Jean-Francois', 'Caron']
>>> d = {'first' : k[0], 'second' : k[1]}
>>> d
{'first': 'Jean-Francois', 'second': 'Caron'}
```

Flow Control

In python, indentation signifies a block or *scope.*

Flow-control structures like **`if`**, **`elif`**, **`else`**, **`while`**, **`for`**, *etc* define a new block.



```
>>> a = 5
>>> if a > 5:
...   print("a is greater than five")
... elif a == 5:
...   print("a is equal to five")
... else:
...   print("a is less than five")
...
a is equal to five
```

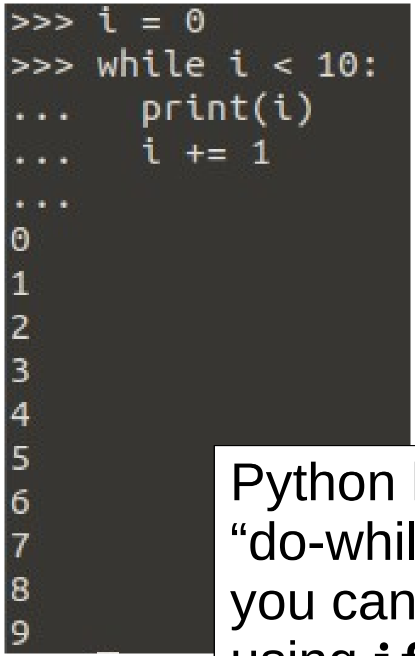
New blocks always start with a colon:

There are 2 spaces here.

`==`, **`!=`** , **`<>`**, **`>`**,**`<=`**, *etc* are comparisons
There are also binary logic operations, notably **`^`** is *exclusive or*, don't confuse it for exponentiation!

- Indentation can be either spaces or tabs, but please **only use spaces.**
- Typically 4 spaces is used, I like using 2.
- It has to be consistent within the structure, but please **be consistent across the whole work.**
- Blocks-within-blocks just indent further.
- You can set most text editors to visibly show "whitespace" to make it easier to see.

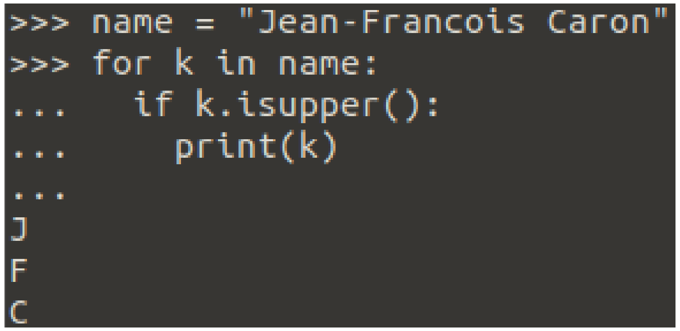
Loops



```
>>> i = 0
>>> while i < 10:
...     print(i)
...     i += 1
...
0
1
2
3
4
5
6
7
8
9
```

`while` loops will repeat the code in the block until the condition is false.

Python has no built-in “do-while” loop, but you can make one using **`if`** and **`while`**.



```
>>> name = "Jean-Francois Caron"
>>> for k in name:
...     if k.isupper():
...         print(k)
...
J
F
C
```

`for` loops will iterate over any sequence (tuple, list, str, dict, *etc*).

n.b.: since dicts are unordered, their iteration order is not guaranteed.

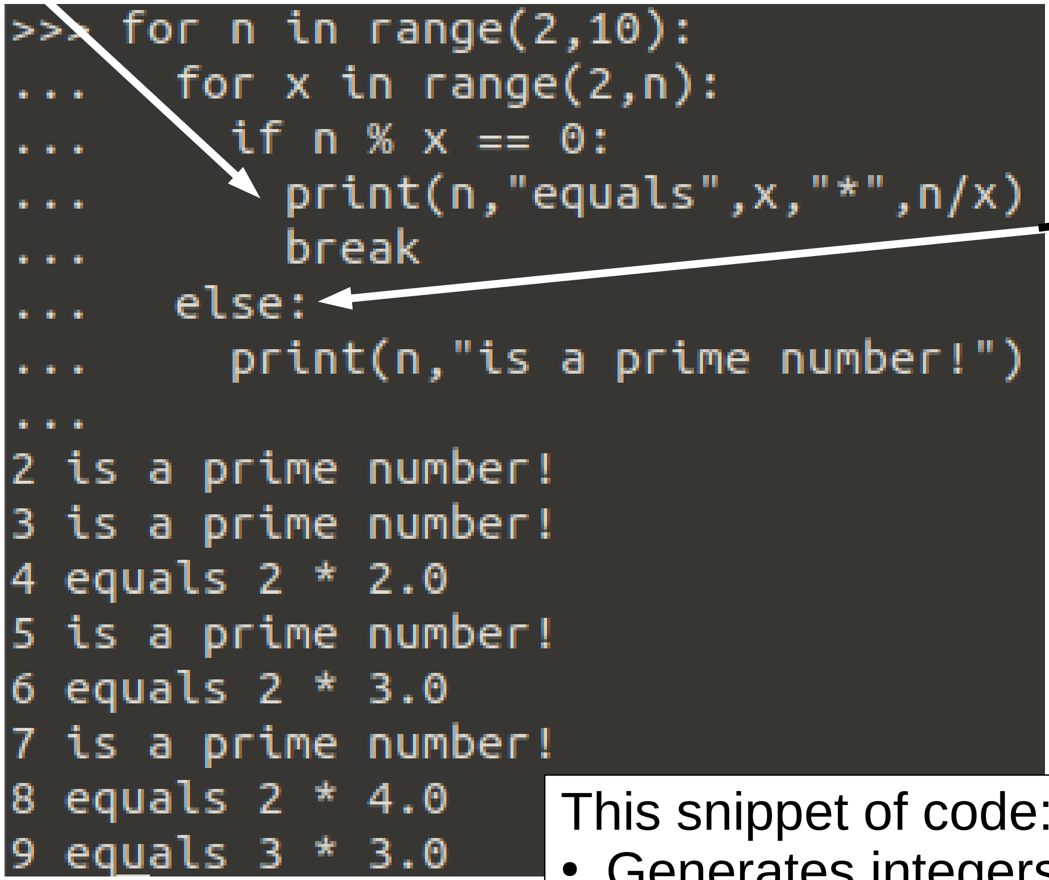
Because of *duck-typing*, many other things are considered sequences, *e.g.* lines in a text file.

`continue` will skip to the next loop iteration.
`break` will break out of the innermost loop.
`else` on a loop starts a new block that executes *after the loop finishes normally.*

this is WEIRD!

Complex Loop Example

Triple-indentation



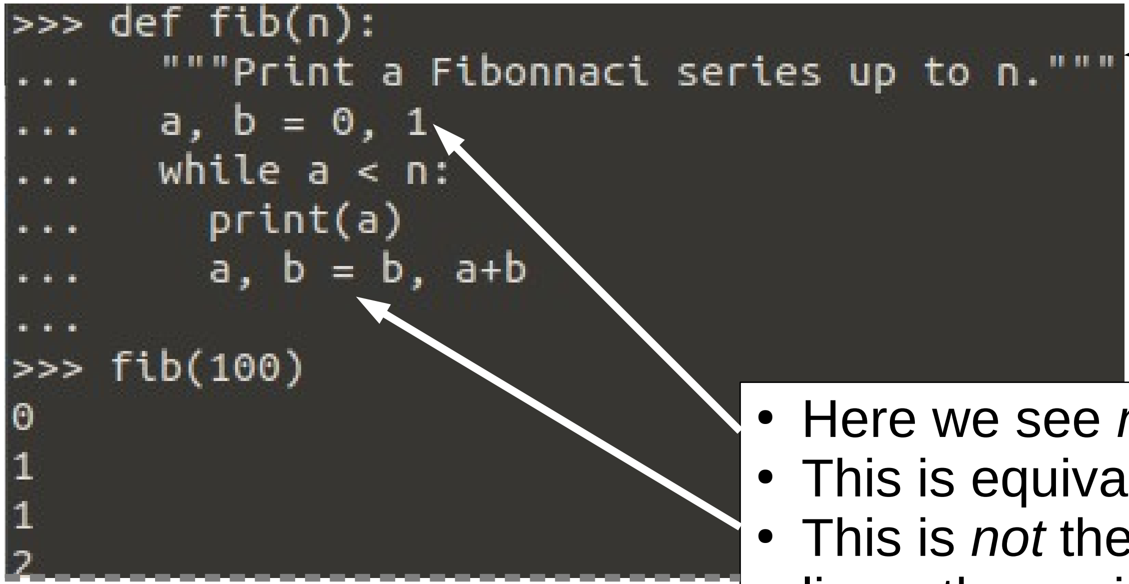
```
>>> for n in range(2,10):
...     for x in range(2,n):
...         if n % x == 0:
...             print(n,"equals",x,"*",n/x)
...             break
...     else:
...         print(n,"is a prime number!")
...
2 is a prime number!
3 is a prime number!
4 equals 2 * 2.0
5 is a prime number!
6 equals 2 * 3.0
7 is a prime number!
8 equals 2 * 4.0
9 equals 3 * 3.0
```

- The **`range(a,b)`** function generates a list of integers from (a,b].
- Recall **`%`** is the modulo operator.
- This **`else`** belongs to the inner **`for`** loop, it executes if the inner loop finishes normally (*i.e.* without **`break`**).

This snippet of code:
- Generates integers n from 2 to 10.
- For each n, generates integers x from 2 to n.
- If x divides n, it prints a message and breaks the inner loop.
- If no x divides n, then n is prime!

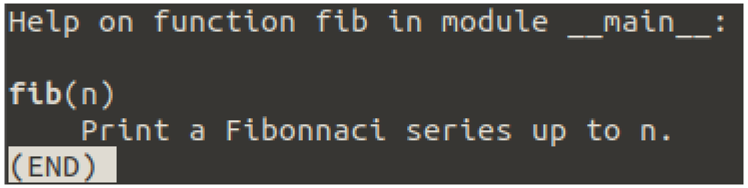
User-Defined Functions

The keyword **def** starts a new block.



```
>>> def fib(n):
...     """Print a Fibonnaci series up to n."""
...     a, b = 0, 1
...     while a < n:
...       print(a)
...       a, b = b, a+b
...
>>> fib(100)
0
1
1
2
```

Triple or sextuple-quotes are used as documentation “docstrings”. So help(fib) gives:

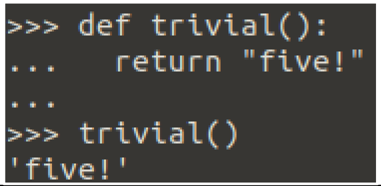


```
Help on function fib in module __main__:

fib(n)
    Print a Fibonnaci series up to n.
(END)
```

- Here we see *multiple-assignment*.
- This is equivalent to **a = 0** and **b = 1** on two lines.
- This is *not* the same as the two assignments on two lines; the assignment happens “at the same time”.
- You can use this to swap two variables without a temporary: **x, y = y, x**
- You can also use this to automatically “unpack” a sequence into individual variables: **i, j = range(2)**

Note: variables defined in a function are *local* to that function. **a** and **b** don’t exist outside of fib.



```
>>> def trivial():
...     return "five!"
...
>>> trivial()
'five!'
```

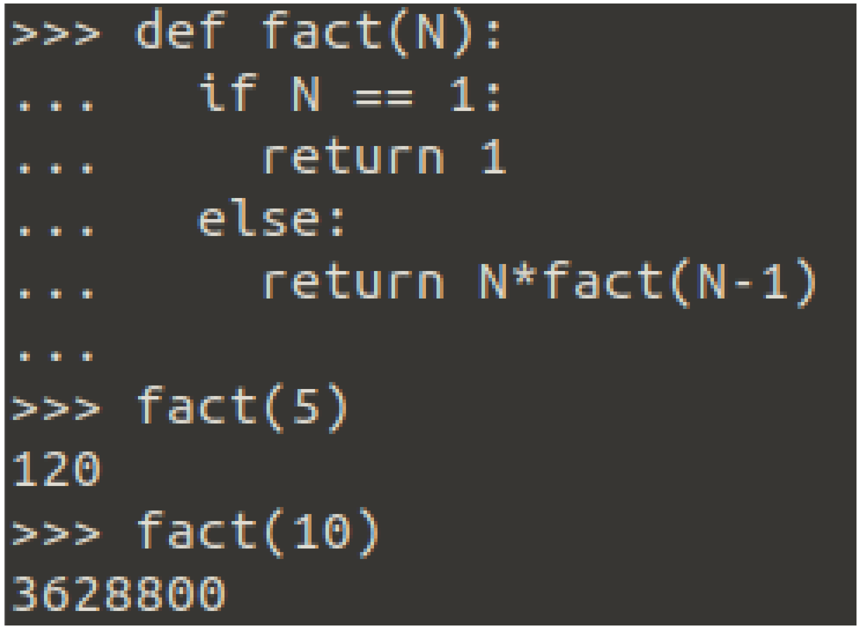
The **return** statement is used to send something out when the function is done. Even without **return**, functions return the special value **None**.

Exercise

- Write a function to calculate and return N! (factorial).
- Tip: You can use recursion (a function that calls itself), but you don’t have to.

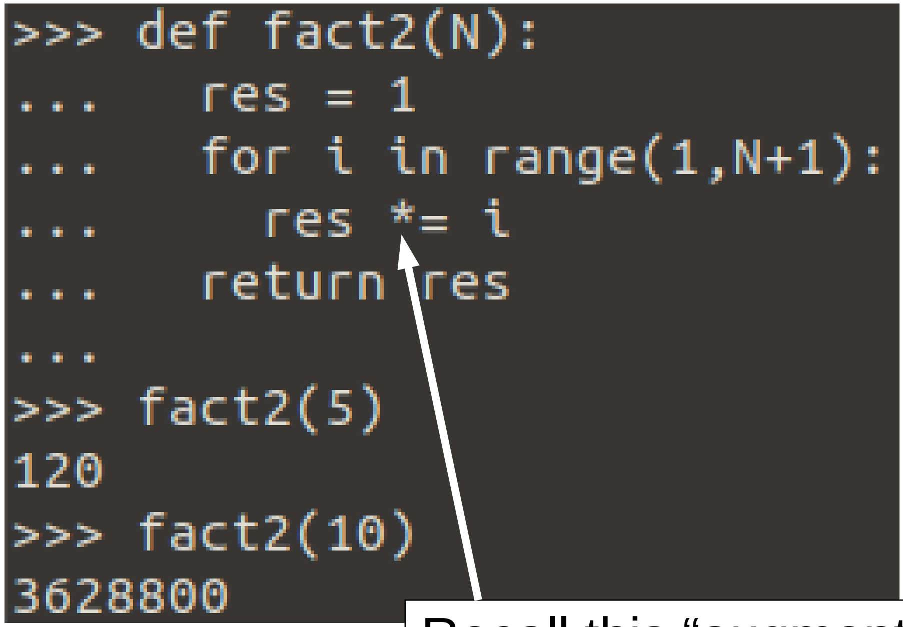
Possible Solutions

Recursive



```
>>> def fact(N):
...     if N == 1:
...         return 1
...     else:
...         return N*fact(N-1)
...
>>> fact(5)
120
>>> fact(10)
3628800
```

Non-recursive



```
>>> def fact2(N):
...     res = 1
...     for i in range(1,N+1):
...         res *= i
...     return res
...
>>> fact2(5)
120
>>> fact2(10)
3628800
```

Recall this "augmented assignment" is equivalent to **`res = res*i`**

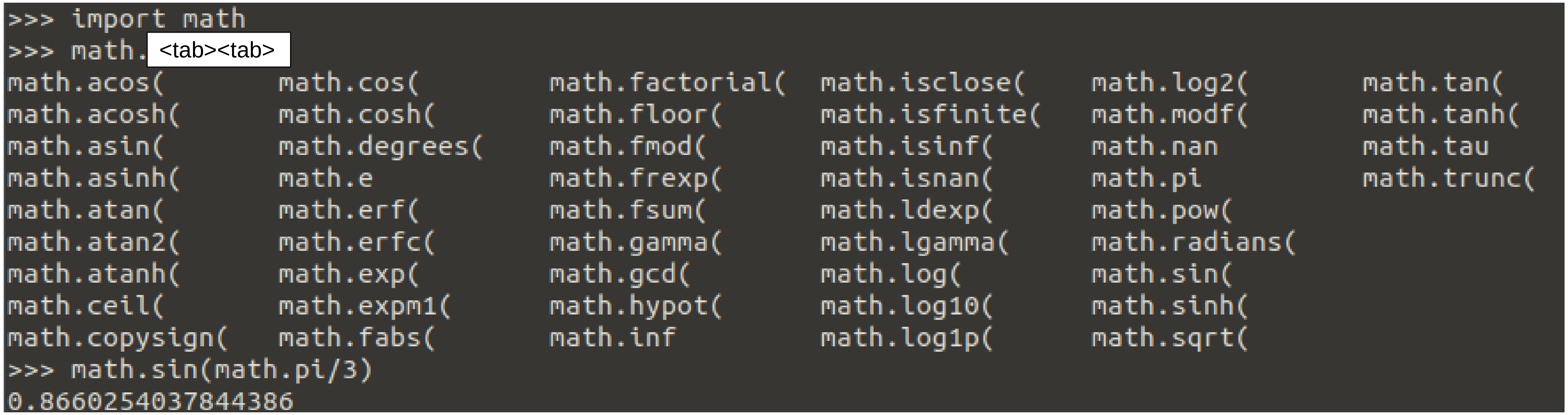
Modules

- Python is “batteries included”, *i.e.* a lot more features are included compared to other languages.
- Most of these features are in standard *modules*.
- There are also 3rd-party non-standard modules (*e.g.* numpy, PyROOT) and you can write your own modules.
- In many cases the solution to a python problem is just finding the right module to do what you want.

Importing Modules

- `import modulename` will import the module.
- If it's not a standard module, there must be a matching **modulename.py** file either in the directory where you started python, or in the list of directories in PYTHONPATH.
  - At the bash prompt, do `echo $PYTHONPATH` or `import sys; print(sys.path)` in python to see the list.
- Please put all your imports at the top of your programs, with one module imported per line.
- You may see `from modulename import *` in examples - **don't do this**, it's bad style.

Module Example & Namespaces



```
>>> import math
>>> math. <tab><tab>
math.acos(        math.cos(         math.factorial(   math.isclose(     math.log2(        math.tan(
math.acosh(       math.cosh(        math.floor(       math.isfinite(    math.modf(        math.tanh(
math.asin(        math.degrees(     math.fmod(        math.isinf(       math.nan          math.tau
math.asinh(       math.e            math.frexp(       math.isnan(       math.pi           math.trunc(
math.atan(        math.erf(         math.fsum(        math.ldexp(       math.pow(
math.atan2(       math.erfc(        math.gamma(       math.lgamma(      math.radians(
math.atanh(       math.exp(         math.gcd(         math.log(         math.sin(
math.ceil(        math.expm1(       math.hypot(       math.log10(       math.sinh(
math.copysign(    math.fabs(        math.inf          math.log1p(       math.sqrt(
>>> math.sin(math.pi/3)
0.8660254037844386
```

- Python has the concept of *namespaces.* A namespace is the "context" in which a name (like a variable or function name) has meaning.
- To access an object inside a namespace, you do **`namespace.objectname`**
- This allows you and module writers to use names without worrying about giving two different objects the same name.
- You can explicitly make new names to "break out": e.g. **`sin = math.sin`**
- For frequently-used names, "breaking out" can reduce the amount of typing required (and a tiny performance boost).
- "Breaking out" makes it harder to understand your code! I almost never do it.
- **`from modulename import *`** "breaks out" ALL the names from the module!

My Favourite Modules

I used **ag --python “import”** to get a big list of all the places where something is imported in my own code.

- ROOT: interaction with ROOT, graphing
- sys: python interpreter internals (e.g. sys.path, sys.argv, sys.version)
- os: interacting with the operating system (os.path, os.mkdir)
- numpy: matrix algebra and numerical computing
- time: timestamps and measuring time
- math: special functions and math
- subprocess: run programs in parallel, launch external programs
- csv: handling comma-separated values files
- random: random number generation
- serial: reading/writing to the serial port (for Arduinos)
- pygame: easily make simple 2D games!

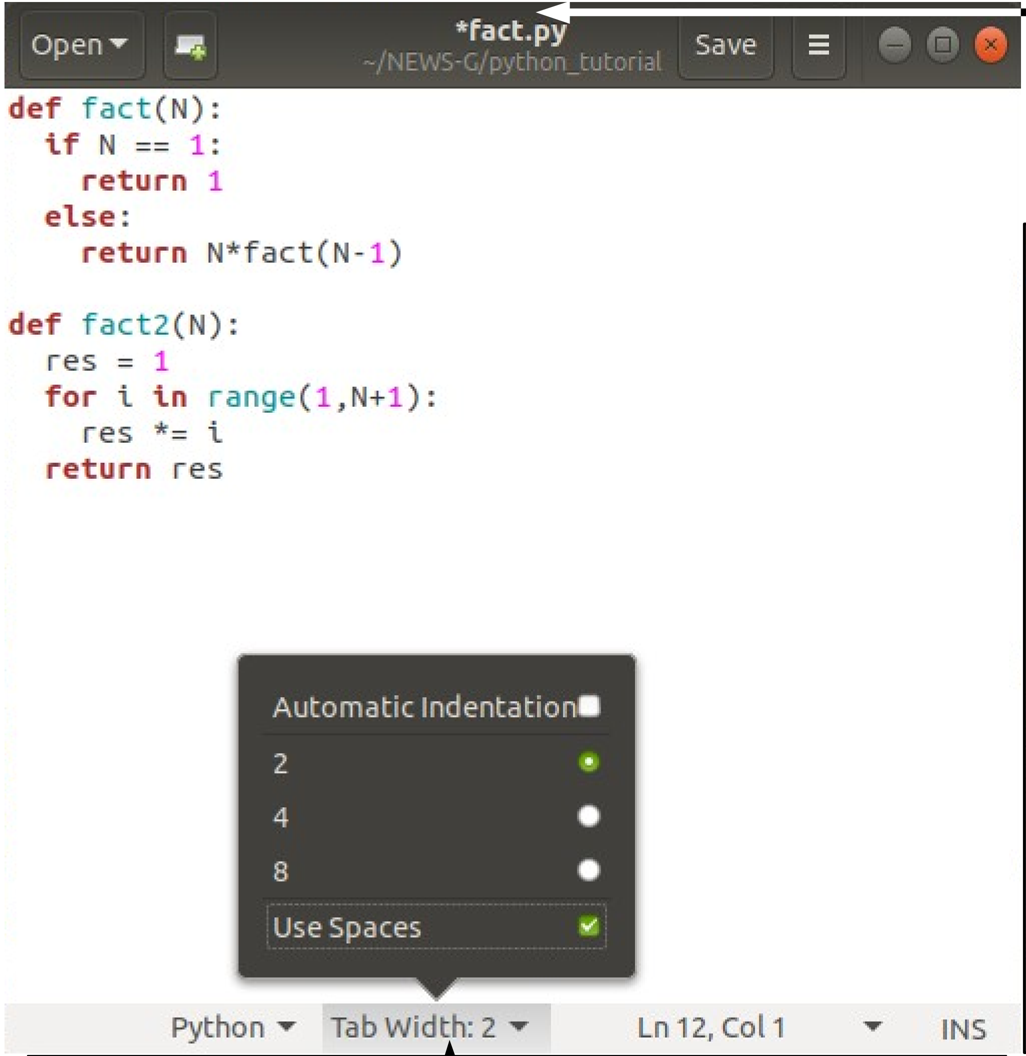
Built-in
3rd-party

Let’s Make A Module!

- Typing functions into the interpreter is tedious.
- Type into a text editor instead!
- Make a directory called “python_tutorial” on your computer and mount it in Docker.

```
sudo docker run -it -v /path/to/python_tutorial/:/home/physuser/python_tutorial:rw soudk/eieioo2021
```

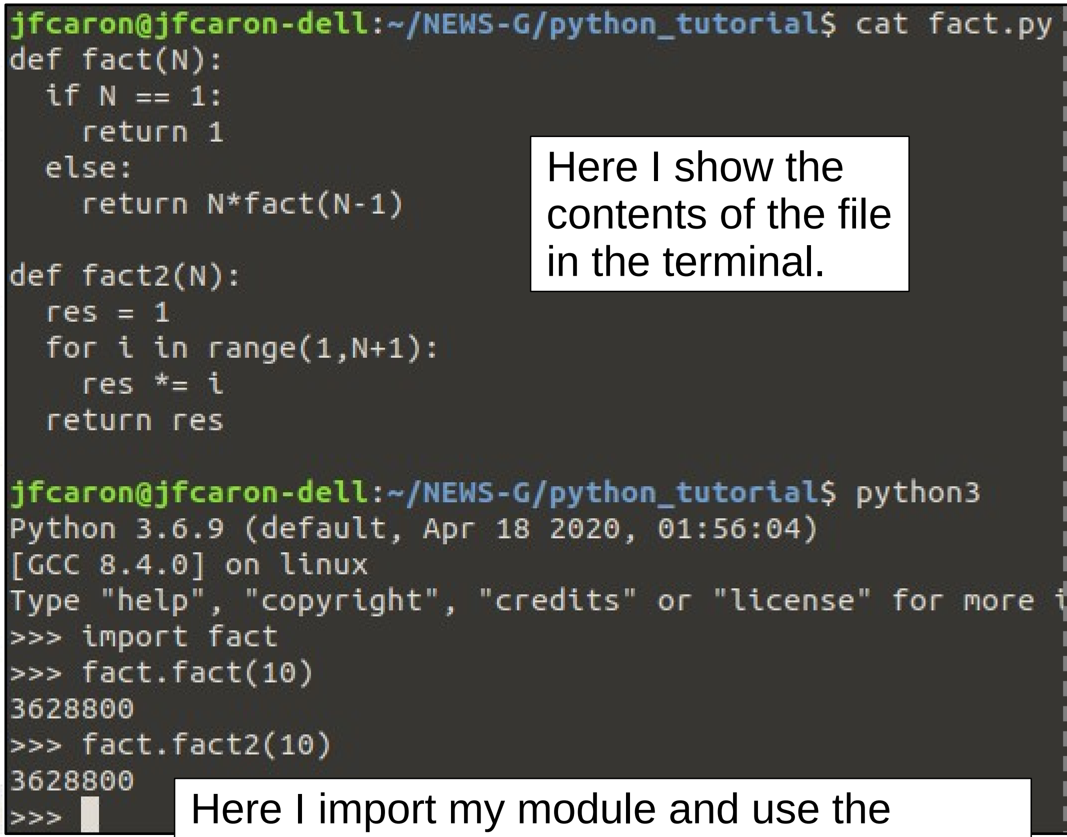
- Open a new file called “fact.py” with your favourite GUI text editor (I suggest gedit).
- Copy & paste your factorial function from your terminal into the file (without the >>> and …).
- Save your module! (easy newbie mistake)



```
def fact(N):
  if N == 1:
    return 1
  else:
    return N*fact(N-1)

def fact2(N):
  res = 1
  for i in range(1,N+1):
    res *= i
  return res
```

Note: the * means I haven't saved my file yet! The file on disk probably empty!



```
jfcaron@jfcaron-dell:~/NEWS-G/python_tutorial$ cat fact.py
def fact(N):
  if N == 1:
    return 1
  else:
    return N*fact(N-1)

def fact2(N):
  res = 1
  for i in range(1,N+1):
    res *= i
  return res

jfcaron@jfcaron-dell:~/NEWS-G/python_tutorial$ python3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more i
>>> import fact
>>> fact.fact(10)
3628800
>>> fact.fact2(10)
3628800
>>> 
```

Here I show the contents of the file in the terminal.

Check/change your indentation settings here. "Use Spaces" makes it so the tab key puts in the right number of spaces.

To make this permanent, go to the Hamburger menu and "Preferences".

Here I import my module and use the functions inside.

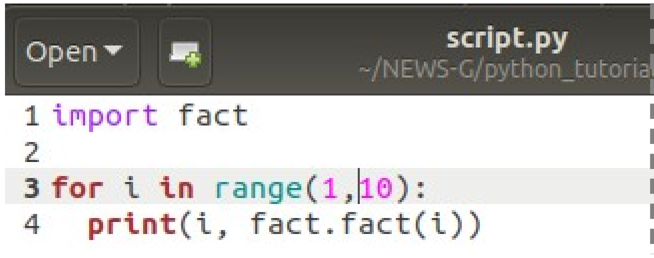
This worked because fact.py is in the same directory where I launched python.

Note that I have to use the namespace!

Tip: Install the gedit-plugin-draw-spaces Ubuntu package to allow gedit to show visible whitespace.

Let’s Write a Script!

- Create a new file called script.py in the same directory.
- In the script, import the module, calculate some factorials and print them.



```python
import fact

for i in range(1,10):
  print(i, fact.fact(i))
```

- At the bash prompt, run the program with:
  python3 script.py
- Now run the script with
  python3 -i script.py

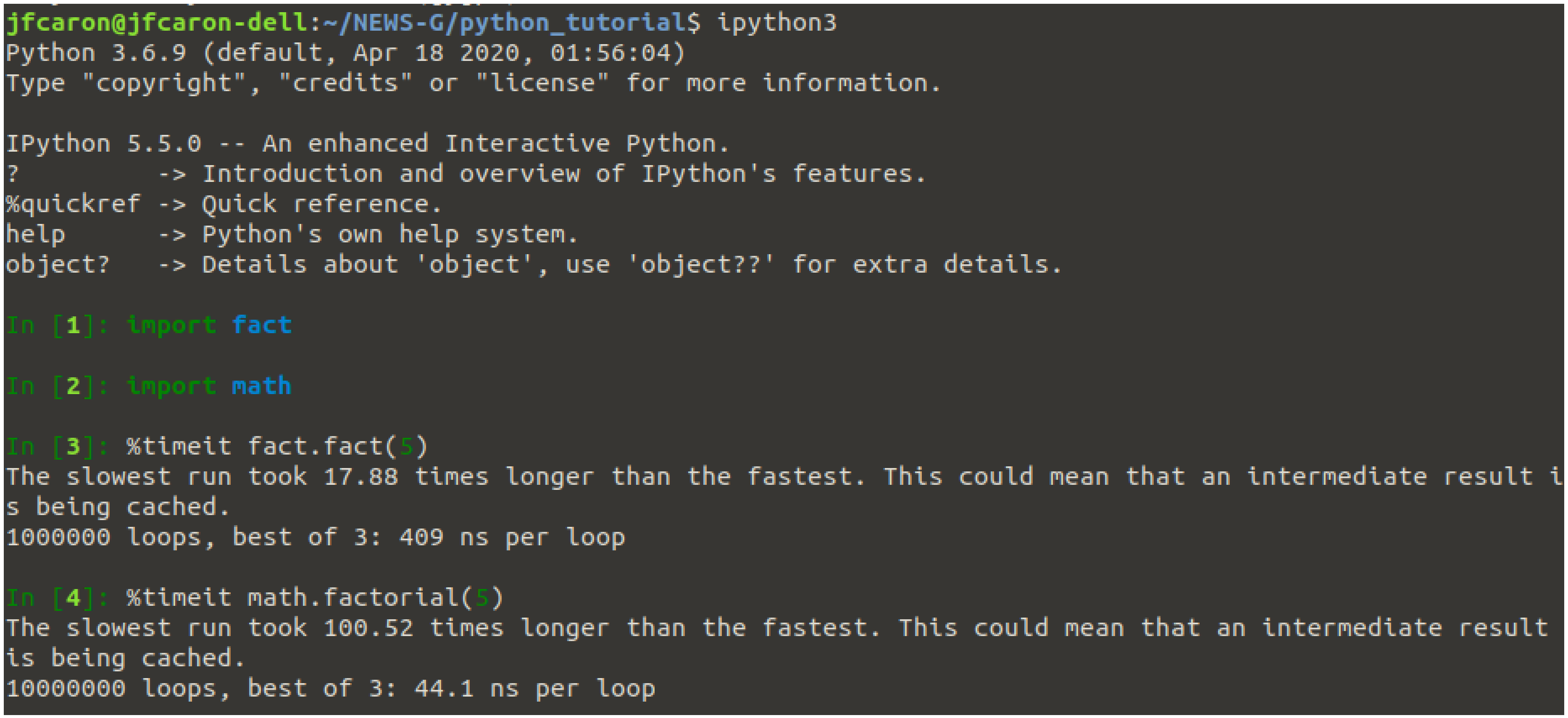
Final Exercise/Homework

- In your script, import the **`time`** and **`math`** built-in modules.
- Use the **`time.time()`** function to calculate how much time it takes to run your factorial function.
- Do the same thing with the **`math.factorial`** function.
- Bonus: Extend the test to time a large number of calculations, or test different arguments to the factorial function.

IPython

- The default python shell is somewhat minimal.
- IPython is a replacement shell that adds a bunch of features. It's implicitly used in Jupyter Notebooks.
  - Auto-identation
  - Up-arrow recalls entire blocks instead of single lines
  - Syntax highlighting as you type
  - Special **`?foo`** syntax to get help (*e.g.* **`?math.sin`**)
  - Special **`%magic`** convenient commands (*e.g.* change directory without leaving python).

IPython Magic Example



```
jfcaron@jfcaron-dell:~/NEWS-G/python_tutorial$ ipython3
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import fact

In [2]: import math

In [3]: %timeit fact.fact(5)
The slowest run took 17.88 times longer than the fastest. This could mean that an intermediate result i
s being cached.
1000000 loops, best of 3: 409 ns per loop

In [4]: %timeit math.factorial(5)
The slowest run took 100.52 times longer than the fastest. This could mean that an intermediate result
is being cached.
10000000 loops, best of 3: 44.1 ns per loop
```

Important Topics Missing

- Default arguments and keyword arguments to functions
- Exception handling with try/except
- List comprehensions and generators
- Basic file input & output, csv files
- Numpy and numerical arrays
- PyROOT (Thursday’s tutorial at 17h00 EST)
- Classes and inheritance

Getting Help

- The official tutorial is quite good: https://docs.python.org/3/tutorial/index.html.
- Use the library and language references on www.python.org.
- Don’t forget about the **`help`** command inside python.
- Search for solutions on the internet!
- Ask in your research group.
- Go to FreeNode on IRC.
  - Go to https://webchat.freenode.net/ or use an IRC client program to connect to chat.freenode.net
  - You will need to register a (free) FreeNode account to join #python
    - Learn to ask questions concisely, specify python 3.
    - #python is very beginner-friendly.
    - Don’t take it personally if people tell you to RTFM.

Beware of old python 2.x examples!
But the main difference is changing
- **`print foo, bar`** to
- **`print(foo, bar)`**

PyHEP 2021 Plug

- PyHEP (Python in High-Energy Physics)
- Lots of topics about 3rd-party modules for statistics, graphing, hardware interfaces, *etc*.
- https://indico.cern.ch/event/1019958/
- Registration is open until July 2nd.
- *No workshop fees*
- 572 participants when I checked.
- I attended in 2020 and it was great!