STATS 507
Data Analysis in Python

Week 1: Syllabus, Installing Python/Jupyter, Data Types, Functions, and Conditionals

Professor Jeffrey Regier

adapted from slides by Keith Levin

Course goals

- Establish a broad background in Python programming
- Prepare you for the inevitable coding interview
- Survey popular tools in academia/industry for data analysis and exploration
- Learn how to read documentation and quickly get familiar with new tools

These tools will be obsolete some day...

...but not your ability to learn new frameworks and solve problems!

Prerequisites

I assume that you have *some* background in programming and statistics

Come speak to me if:

- this is your first programming course
- you have never taken a probability or statistics course

This course is probably not for you if:

- you have no programming background

MORE ON THIS LATER.

Course structure

Part 1: Introduction to Python

Data types, functions, classes, objects, functional programming

Part 2: Numerical Computing and Data Visualization

numpy, scipy, scikit-learn, matplotlib, Seaborn

Part 3: Dealing with structured data

pandas, regular expressions, retrieving web data, SQL, real datasets

Part 4: Deep learning

PyTorch, SGD, Multi-layer perceptrons, regularization, ConvNets, Kaggle

Instructors

Jeffrey Regier, regier@umich.edu

a.k.a. “Jeff” ; a.k.a. “Professor Re-Gear” ♻ ⚙

My office hours

Thursdays, 4 pm -- 5:30 pm

and by appointment

Graduate Student Instructors

Derek Hansen, dereklh@umich.edu

Statistics PhD student; Python guru

Brian Manzo, bmanzo@umich.edu

Statistics PhD student; Python guru & 507 alum

GSI office hours

Mondays, 11:30 am -- 1 pm

Tuesdays, 9:00 am -- 10:30 am

Course website

https://umich.instructure.com/courses/417299/

- Read the syllabus
- Look ahead at lecture slides
- Download the homework assignments
- Ask questions about homework assignments
- Find office hours times and Zoom links
- Submit your homework solutions
- Check your grades
- and more!

Getting help

	Canvas discussion board	GSI office hours	my office hours	email GSIs	email me
questions about homework	☺ *	☺		✗	✗
questions about lecture / slides / course topics		☺	😃	✗	✗
questions / concerns about grading	✗	☺	☺ **	☺	
personal matters; concerns about course; extended illness	✗		✓	✗	✓

* For questions about homework that cannot be asked without revealing a solution, please ask during GSI office hours rather than through Canvas.

** Please ask the GSIs first about homework grading; come to me if your concern is not resolved.

507 is a **synchronous** course this semester

All lectures will be recorded and I don’t take attendance.

But, if you are in a very different time zone,

1) **it may be difficult to attend GSI office hours;** you may have to get through the homework largely without support.

2) you may have to take the final exam at an unusual time.

Textbooks

The first part of the course is based on *Python for Everybody* by Charles Severance:

https://www.py4e.com/book.php

The last part of the course is based on *Dive into Deep Learning:*

http://d2l.ai/

Online resources

It is a goal of this course to get you comfortable reading docs!
Read and understand what you can, google terms you don’t understand.

- NumPy quickstart, https://numpy.org/devdocs/user/quickstart.html
- SciPy tutorial, https://docs.scipy.org/doc/scipy/reference/tutorial/
- Pyplot tutorial, https://matplotlib.org/tutorials/introductory/pyplot.html
- Pandas user guide https://pandas.pydata.org/pandas-docs/stable/user_guide/
- Seaborn tutorial https://seaborn.pydata.org/tutorial.html
- scikit-learn tutorial https://scikit-learn.org/stable/tutorial/
- PyTorch tutorials, https://pytorch.org/tutorials/

Grading

Final grades will be based on

- ~weekly homework (70%)
- a final exam (30%)

The distribution of final grades is expected to be similar to what it has been for previous offerings of STATS 507:

45% A/A+,   70% A-/A/A+,   98% B- or above.

Homework (70%)

Students enter 507 with a wide variety of programming backgrounds.

The course workload is heavy for some students and light for others.

1. Students with the strongest programming backgrounds may complete each weekly homework assignment in as little as **5 hours**.
2. Many students complete the weekly homeworks in around **10 hours**.
3. Students with little prior programming experience will likely learn the most; however, they may need to devote as much as **25 hours weekly** to completing the homework assignments!

Homework & Late Days

Homework due dates are strict. However,

- You have seven “late days” to use over the course of the semester.
- For each late day you spend, you extend the deadline of a homework by 24 hours.
- You may spend multiple late days per homework.
- Once you have turned in your homework you may not spend more late days to turn in your homework again.

The purpose of this late day policy is to enable you to deal with unexpected circumstances (e.g., illness, family emergencies, job interviews) without having to come to me. Of course, if dire circumstances arise (e.g., long-term illness that causes you to miss multiple weeks of lecture), please speak with me as promptly as possible.

Don’t plagiarize!

- You may discuss homeworks with your fellow students, but the work you submit must be your own.
- You may not share your homework solutions notebook with classmates, and you may not access other students’ solutions.
- You must disclose in your homework whom (if anyone) you discussed the homework with.
- Submissions will be checked with the MOSS Plagiarism Detector.
- A zero grade will be given for submissions that contain any plagiarized code.
- All incidents of academic dishonesty will be reported to Rackham, which typically administers additional penalties upon conviction.

Final Exam (30%)

During the first half of the course, students are expected to learn Python well enough that they can efficiently solve basic programming problems. Students with these skills are well prepared for technical interviews.

The final exam will test that students have acquired these skills. Students will be provided with ample practice problems ahead of time (in addition to homework problems). Practice problems may be found at https://hackerrank.com and https://codechef.com too.

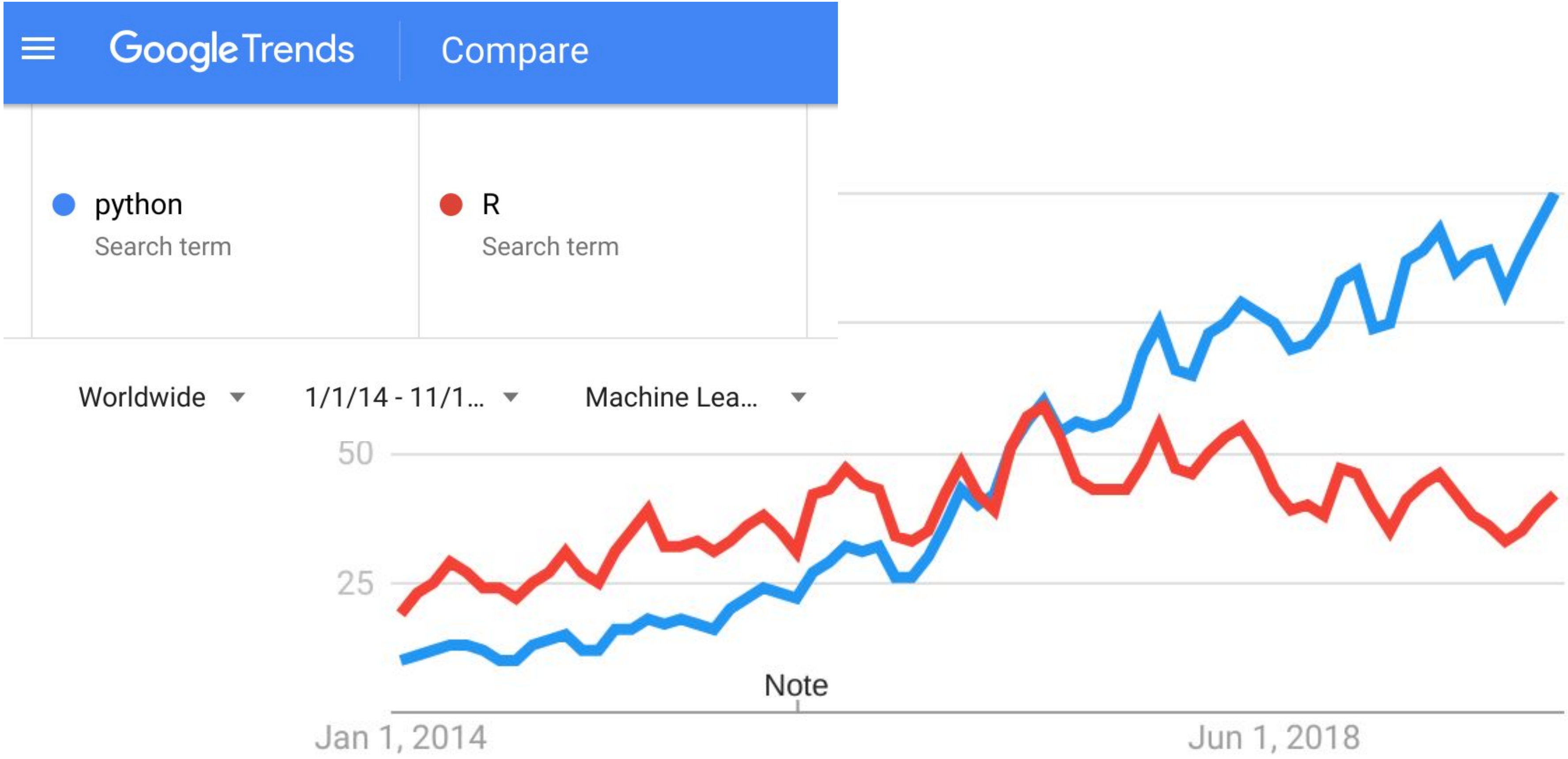
The final exam will be in class on **Friday, April 16, 2021 from 2:00 pm until 4:00 pm (eastern).**

An alternative final exam time will offered **only** for international students in a very different time zone (Asia). This international final exam will be on **Friday, April 16, 2021 from 8:00 pm to 10:00 pm (eastern).**

The final exam will not be given at other times.

Lesson 1: Introduction to Python

Increasingly, Python is the language of data science

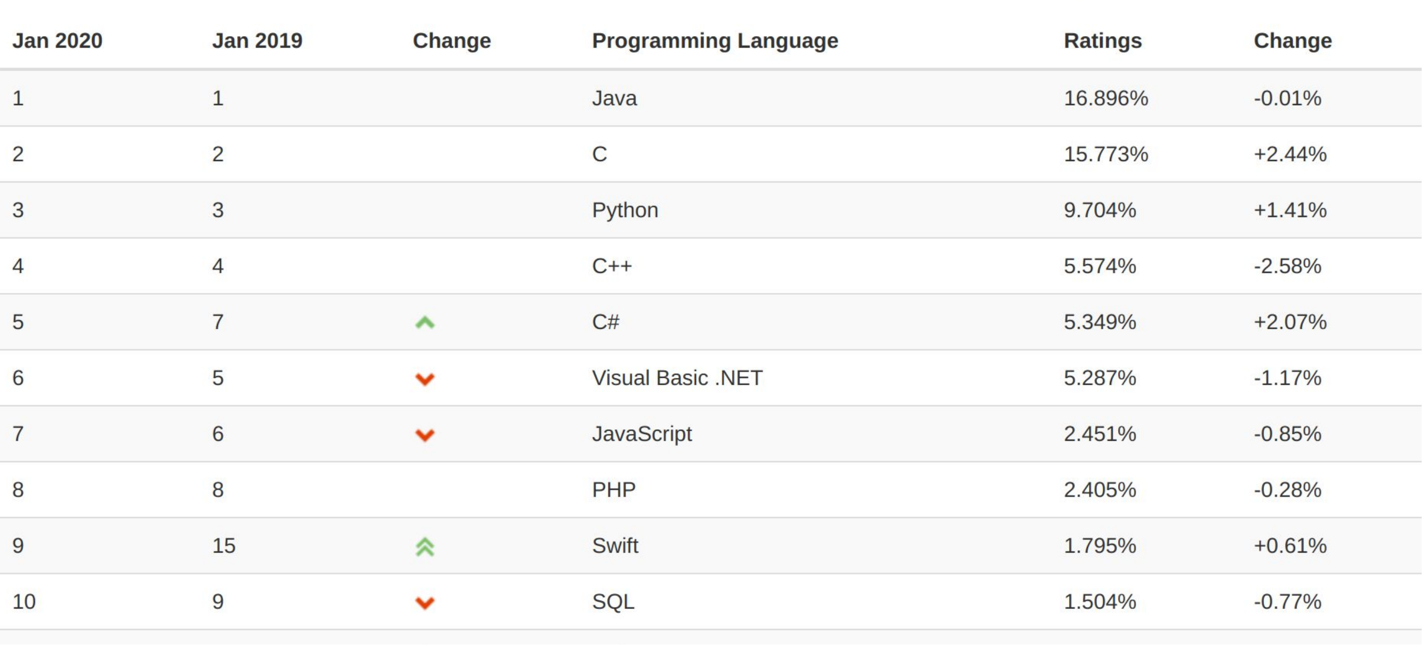


Worldwide ▾

1/1/14 - 11/1… ▾

Machine Lea… ▾

Python is gaining popularity as general purpose programming language too



Jan 2020	Jan 2019	Change	Programming Language	Ratings	Change
1	1		Java	16.896%	-0.01%
2	2		C	15.773%	+2.44%
3	3		Python	9.704%	+1.41%
4	4		C++	5.574%	-2.58%
5	7	▲	C#	5.349%	+2.07%
6	5	▼	Visual Basic .NET	5.287%	-1.17%
7	6	▼	JavaScript	2.451%	-0.85%
8	8		PHP	2.405%	-0.28%
9	15	▲▲	Swift	1.795%	+0.61%
10	9	▼	SQL	1.504%	-0.77%

source: tiobe.com

Python: Overview

Python is a **dynamically typed**, **interpreted** programming language
Created by Guido van Rossum in 1991
Maintained by the Python Software Foundation

Design philosophy: simple, readable code

Python syntax differs from R, Java, C/C++, MATLAB
whitespace delimited
limited use of brackets, semicolons, etc

Image credit: https://www.python.org/community/logos/

Python: Overview

Python is a **dynamically typed**, **interpreted** programming language

Created by Guido van Rossum in 1991

Maintained by the Python Software Foundation

Design philosophy: simple, readable code

Python syntax differs from R, Java, C/C++, MATLAB

whitespace delimited

limited use of brackets, semicolons, etc

In many languages, when you declare a variable, you must specify the variable’s **type** (e.g., int, double, Boolean, string). Python does not require this.

Image credit: https://www.python.org/community/logos/

Python: Overview

Python is a **dynamically typed** **interpreted** programming language

- Created by Guido van Rossum in 1991
- Maintained by the Python Software Foundation

Design philosophy: simple, readable code

Python syntax differs from R, Java, C/C++, MATLAB

- whitespace delimited
- limited use of brackets, semicolons, etc

Some languages (e.g., C/C++ and Java) are **compiled**: we write code, from which we get a runnable program via **compilation**. In contrast, Python is **interpreted**: A program, called the **interpreter**, runs our code directly, line by line.

Compiled vs interpreted languages: compiled languages are (generally) faster than interpreted languages, typically at the cost of being more complicated.

Image credit: https://www.python.org/community/logos/

Running Python

Several options for running Python

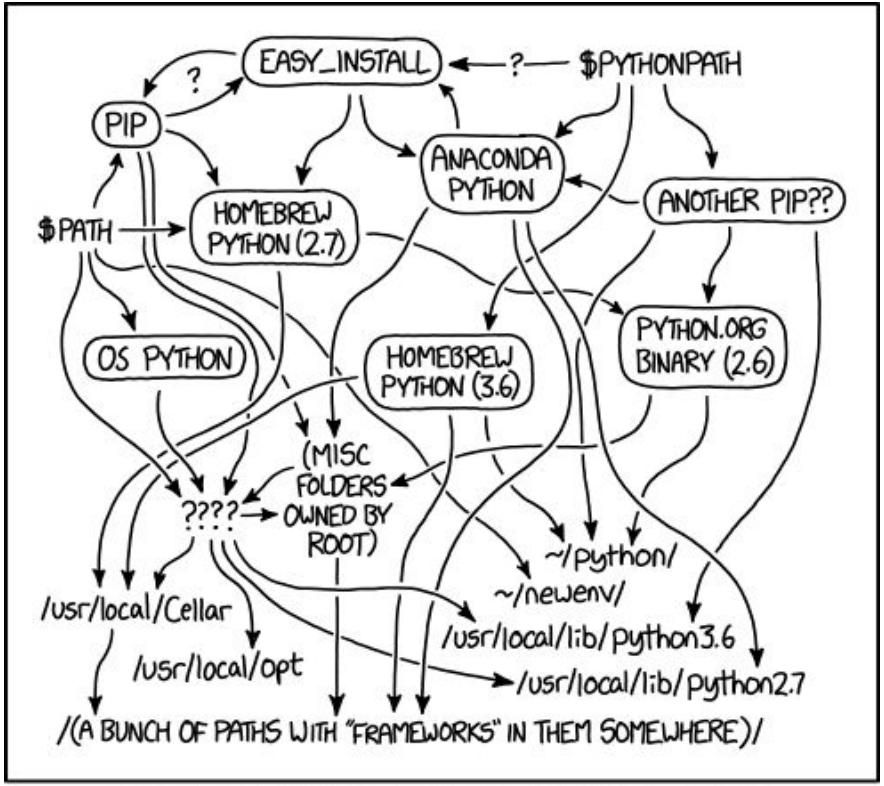
- locally installed Python interpreter
- Jupyter: https://jupyter.org/
- PythonAnywhere: https://www.pythonanywhere.com/
- Google colab: https://colab.research.google.com/

Your homeworks must be handed in as Jupyter notebooks

- But you should also be comfortable with the interpreter and running Python on the command line
- Installing Jupyter: https://jupyter.readthedocs.io/en/latest/install.html
    - **Note:** Jupyter recommends Anaconda: https://www.anaconda.com/
        - But it's your choice whether to use Anaconda or Python.org + pip.

Image credit: https://www.python.org/community/logos/

Installing Python



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Let’s install Python and Jupyter

Python:

https://www.python.org/downloads/release/python-387/

or `conda create -n stats507 python=3.8`

`followed by`

```
conda activate stats507
```

Jupyter:

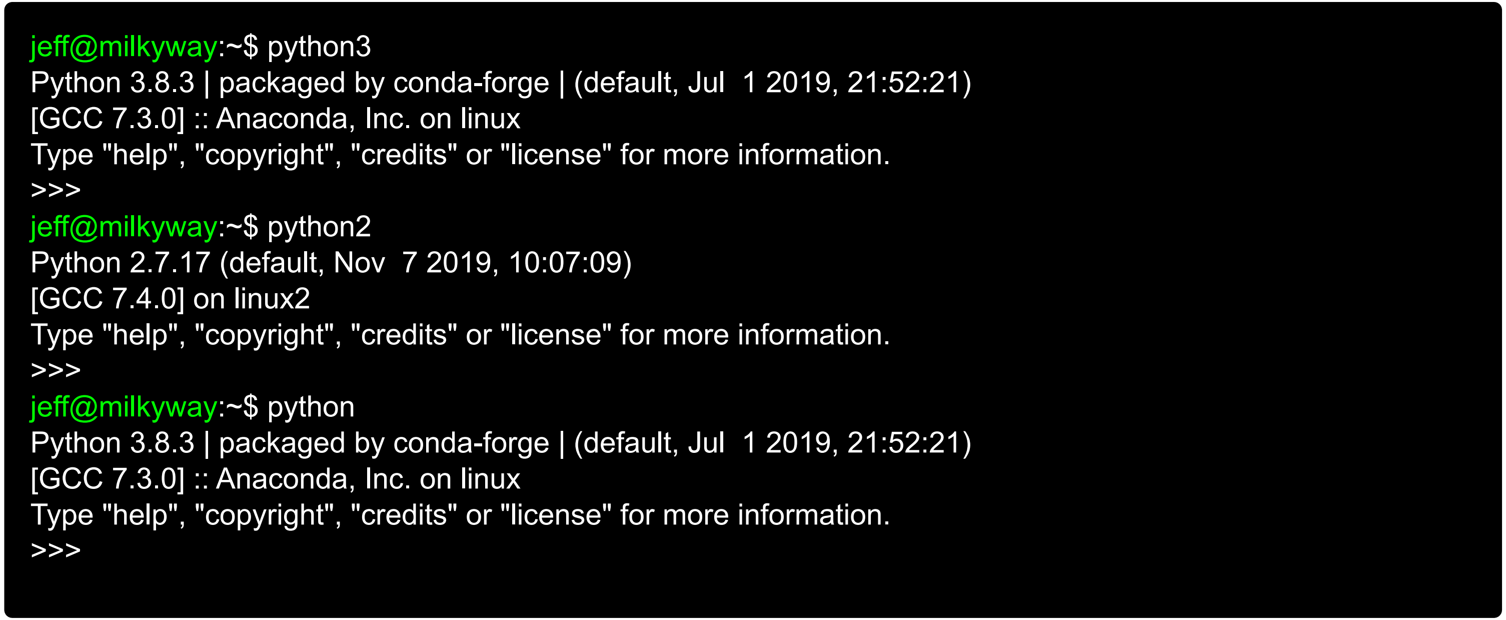
https://jupyter.org/install

In summary, 1) open terminal (mac) or command prompt (windows), and

2) type `pip3 install notebook`

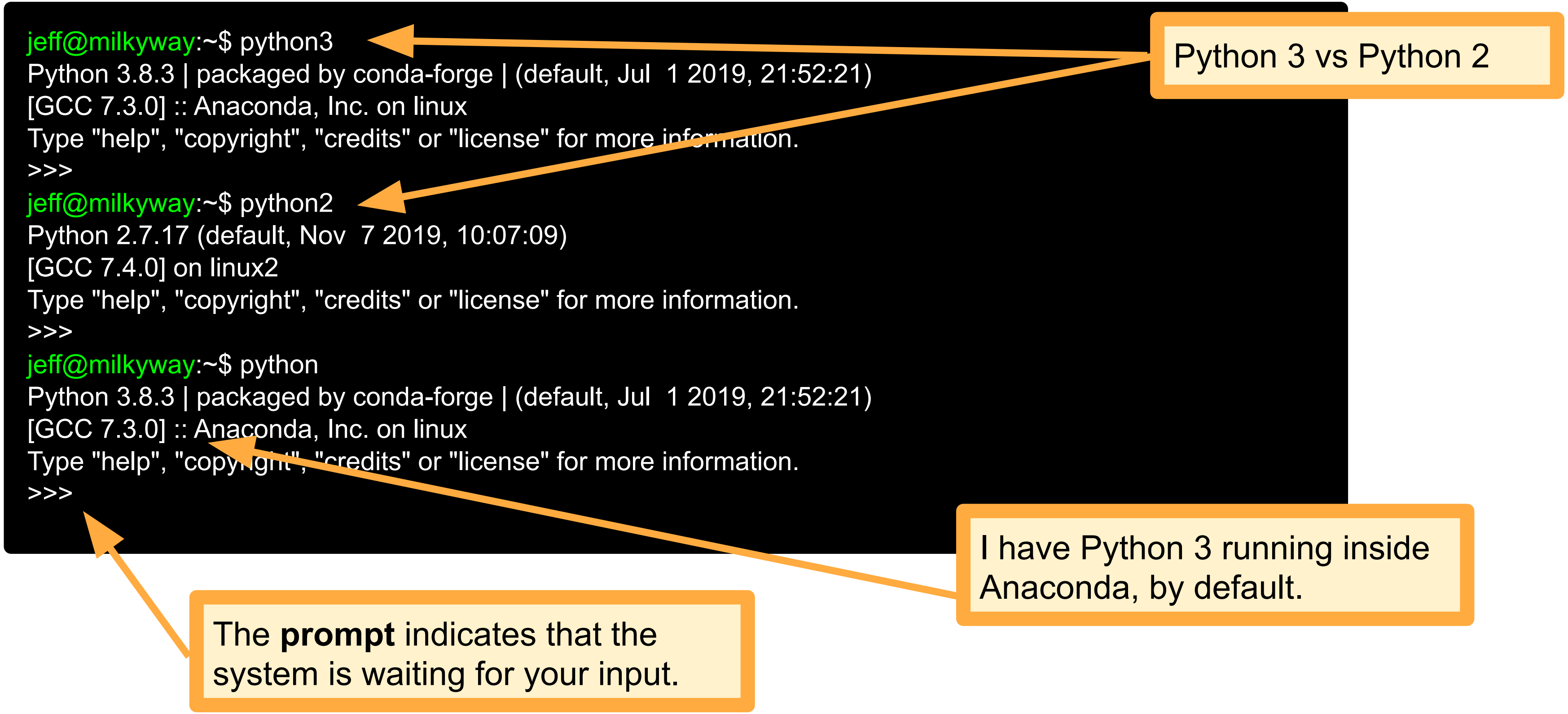
or `conda install jupyter`

Python Interpreter on the Command Line



```
jeff@milkyway:~$ python3
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python2
Python 2.7.17 (default, Nov  7 2019, 10:07:09)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python Interpreter on the Command Line



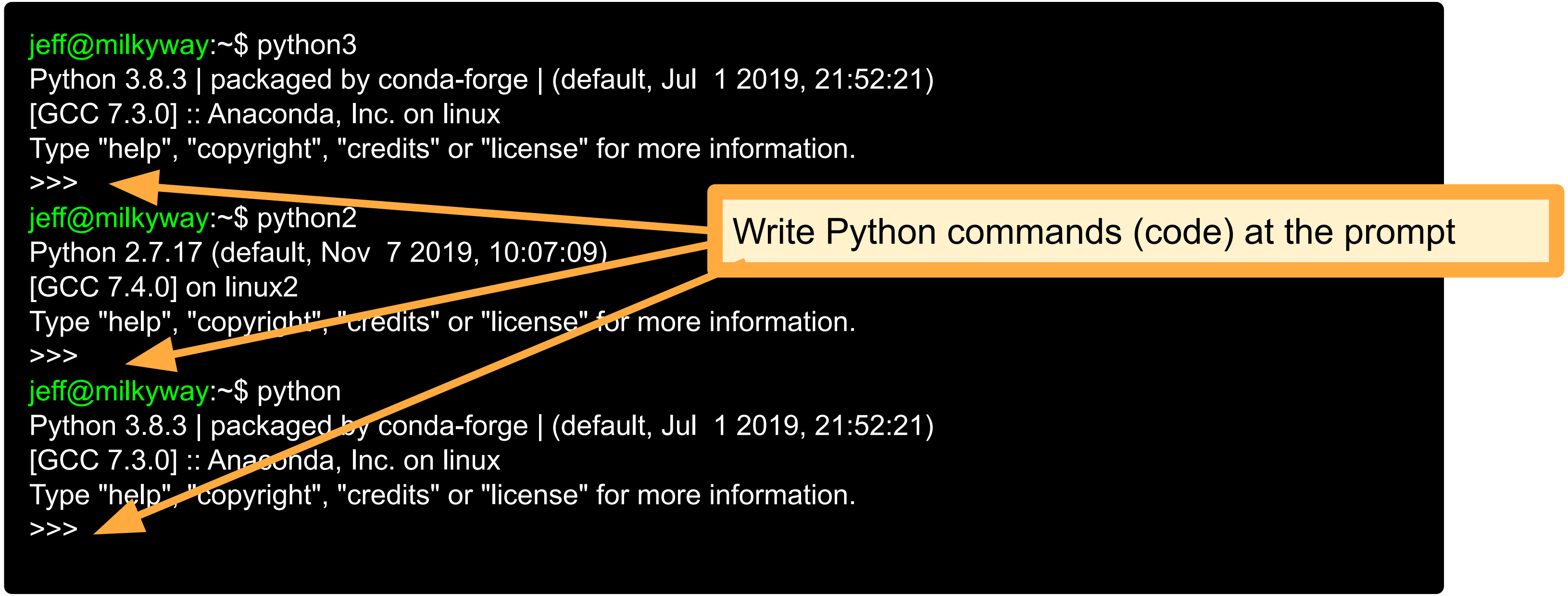
```
jeff@milkyway:~$ python3
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python2
Python 2.7.17 (default, Nov  7 2019, 10:07:09)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python 3 vs Python 2

I have Python 3 running inside Anaconda, by default.

The **prompt** indicates that the system is waiting for your input.

Python Interpreter on the Command Line



```
jeff@milkyway:~$ python3
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python2
Python 2.7.17 (default, Nov  7 2019, 10:07:09)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
jeff@milkyway:~$ python
Python 3.8.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Write Python commands (code) at the prompt

Python in Jupyter

Creates “notebook files” for running **Ju**lia, **Py**thon and **R**

Example notebook:

https://nbviewer.jupyter.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb

Clean, well-organized presentation of code, text and images, in one document

Installation: https://jupyter.readthedocs.io/en/latest/install.html

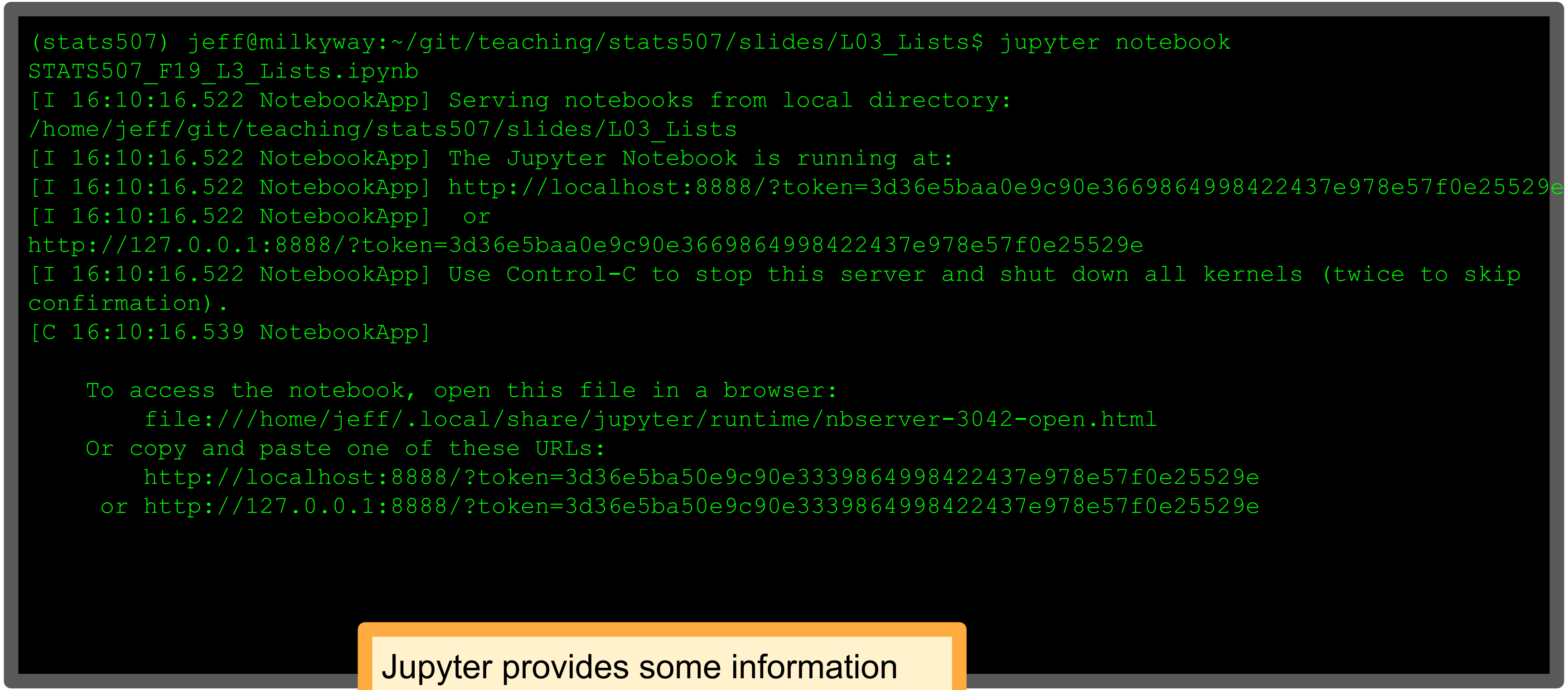
Documentation on running: https://jupyter.readthedocs.io/en/latest/running.html

Good tutorials:

https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook

https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html

Running Jupyter

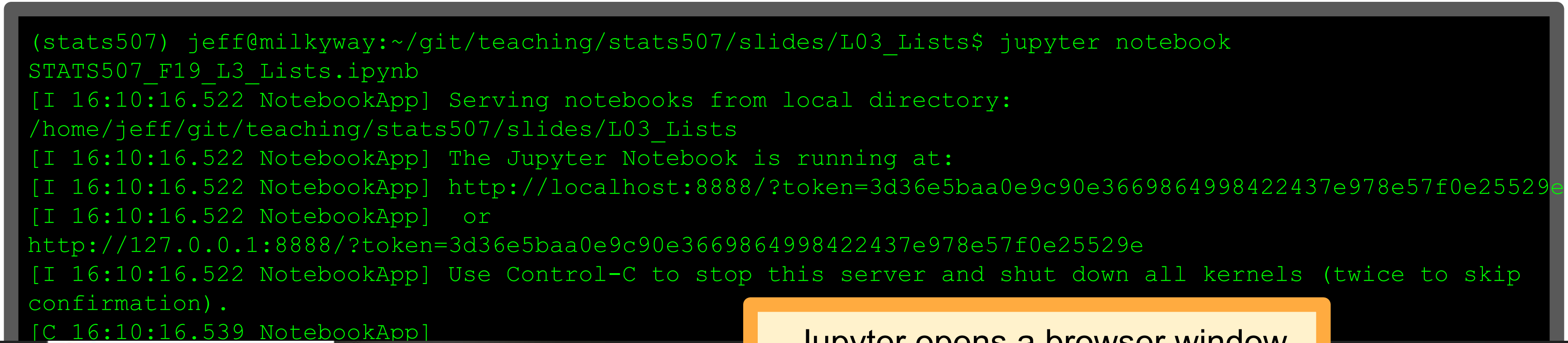


```
(stats507) jeff@milkyway:~/git/teaching/stats507/slides/L03_Lists$ jupyter notebook
STATS507_F19_L3_Lists.ipynb
[I 16:10:16.522 NotebookApp] Serving notebooks from local directory:
/home/jeff/git/teaching/stats507/slides/L03_Lists
[I 16:10:16.522 NotebookApp] The Jupyter Notebook is running at:
[I 16:10:16.522 NotebookApp] http://localhost:8888/?token=3d36e5baa0e9c90e366986499842243 7e978e57f0e25529e
[I 16:10:16.522 NotebookApp]  or
http://127.0.0.1:8888/?token=3d36e5baa0e9c90e3669864998422437e978e57f0e25529e
[I 16:10:16.522 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip
confirmation).
[C 16:10:16.539 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/jeff/.local/share/jupyter/runtime/nbserver-3042-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3d36e5ba50e9c90e3339864998422437e978e57f0e25529e
     or http://127.0.0.1:8888/?token=3d36e5ba50e9c90e3339864998422437e978e57f0e25529e
```

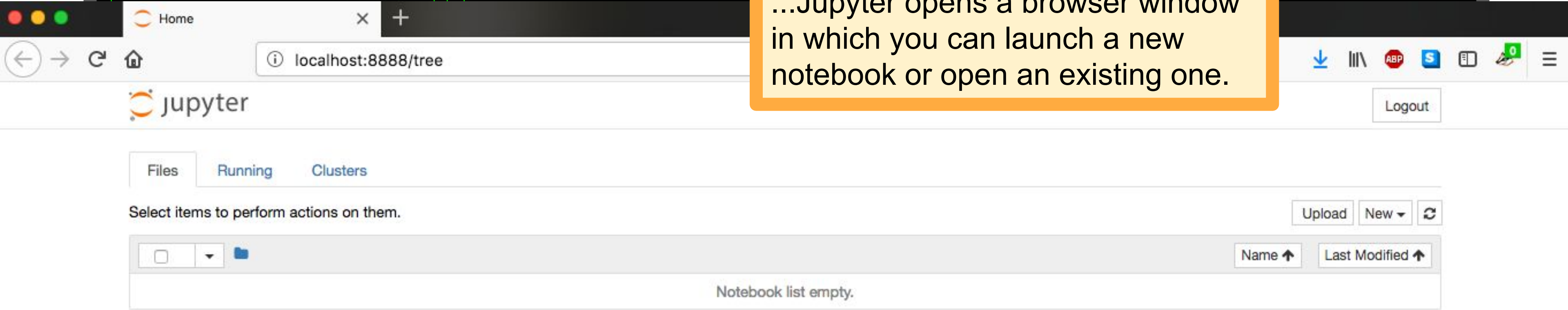
Jupyter provides some information about its startup process, and then...

Running Jupyter



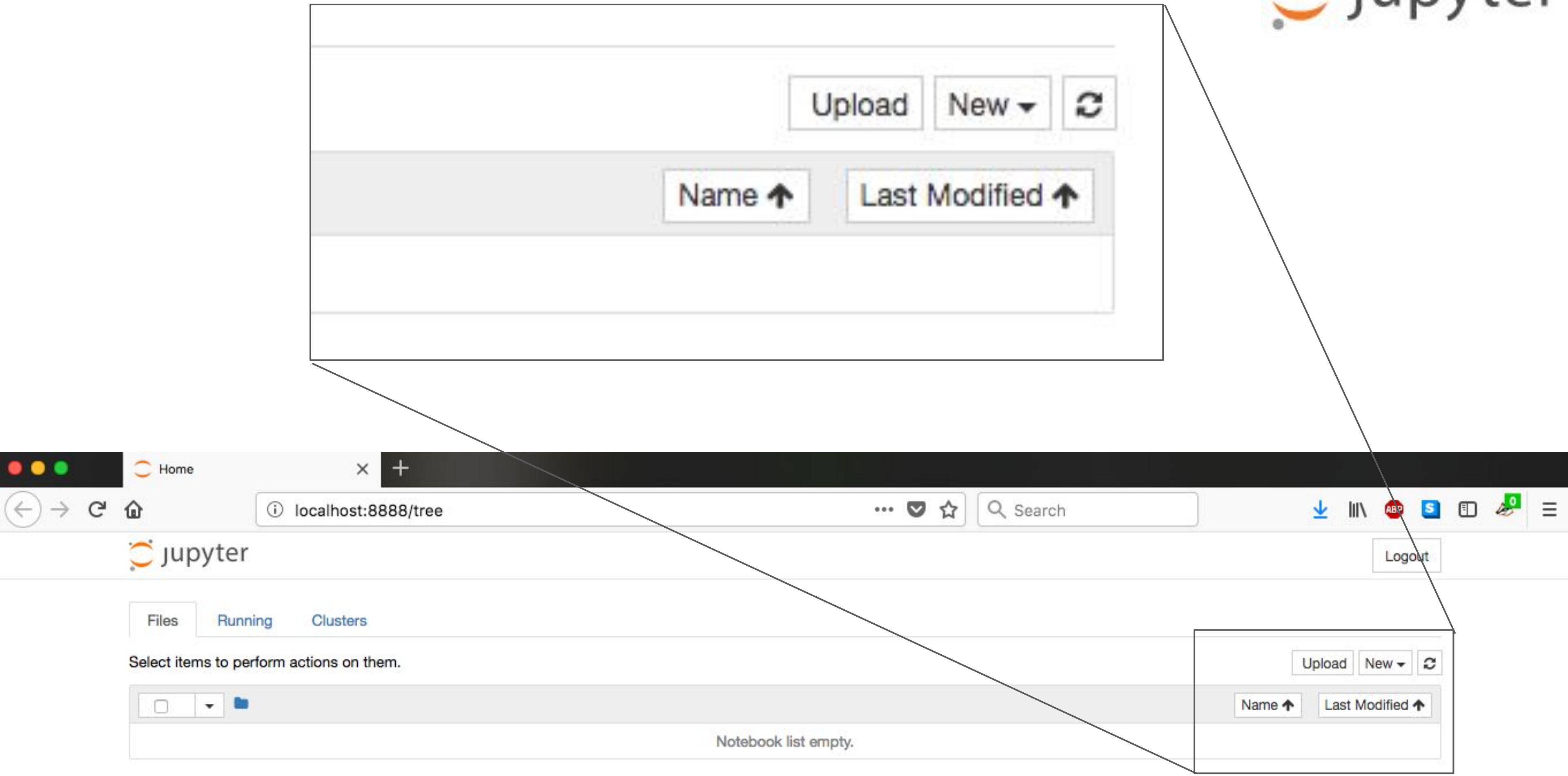
```
(stats507) jeff@milkyway:~/git/teaching/stats507/slides/L03_Lists$ jupyter notebook
STATS507_F19_L3_Lists.ipynb
[I 16:10:16.522 NotebookApp] Serving notebooks from local directory:
/home/jeff/git/teaching/stats507/slides/L03_Lists
[I 16:10:16.522 NotebookApp] The Jupyter Notebook is running at:
[I 16:10:16.522 NotebookApp] http://localhost:8888/?token=3d36e5baa0e9c90e3669864998422437e978e57f0e25529e
[I 16:10:16.522 NotebookApp]  or
http://127.0.0.1:8888/?token=3d36e5baa0e9c90e3669864998422437e978e57f0e25529e
[I 16:10:16.522 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip
confirmation).
[C 16:10:16.539 NotebookApp]
```

...Jupyter opens a browser window in which you can launch a new notebook or open an existing one.



Select items to perform actions on them.

Notebook list empty.



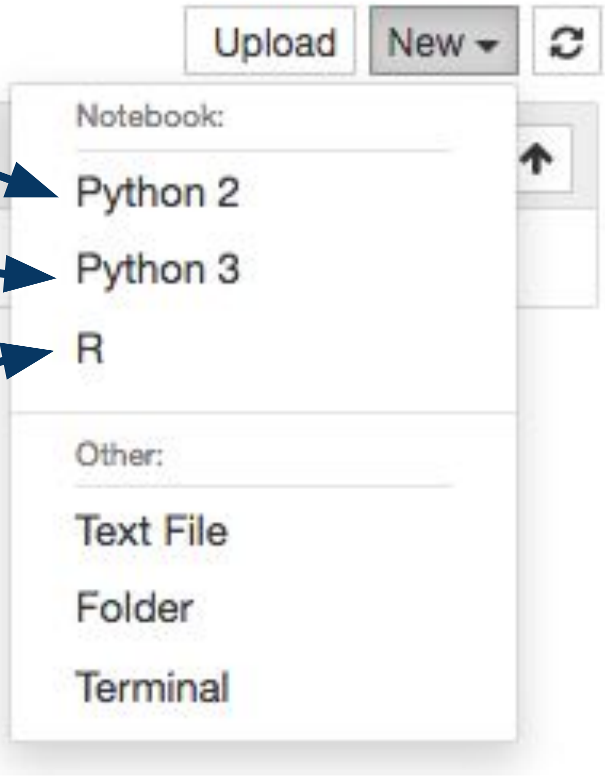
Select items to perform actions on them.

Notebook list empty.

Creates a new notebook file running Python 2.

Creates a new notebook file running Python 3.

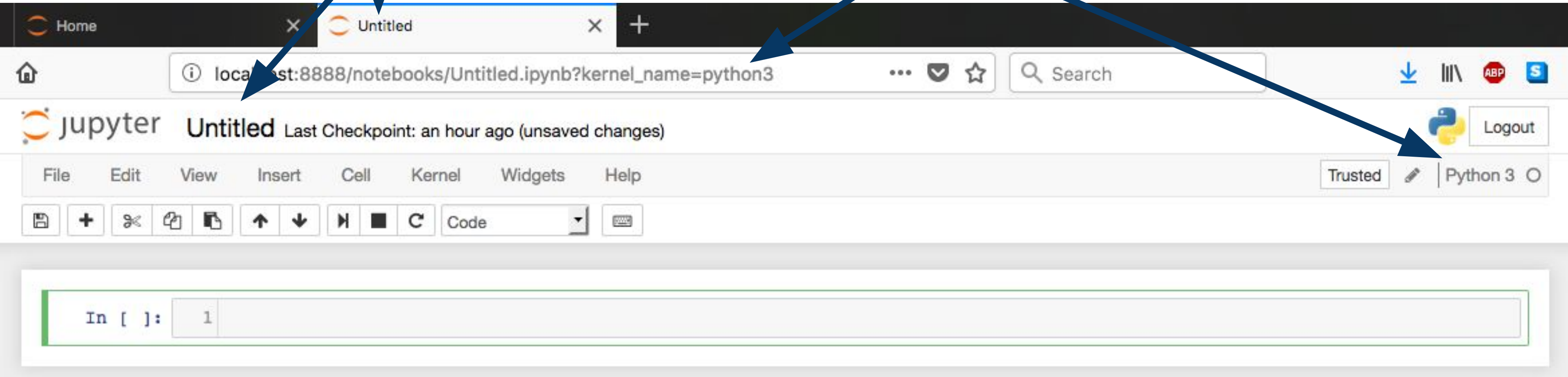
Creates a new notebook file running R.



Note: Jupyter can also run other programming languages, such as Julia, if they are installed.

Notebook doesn't have a title, yet.

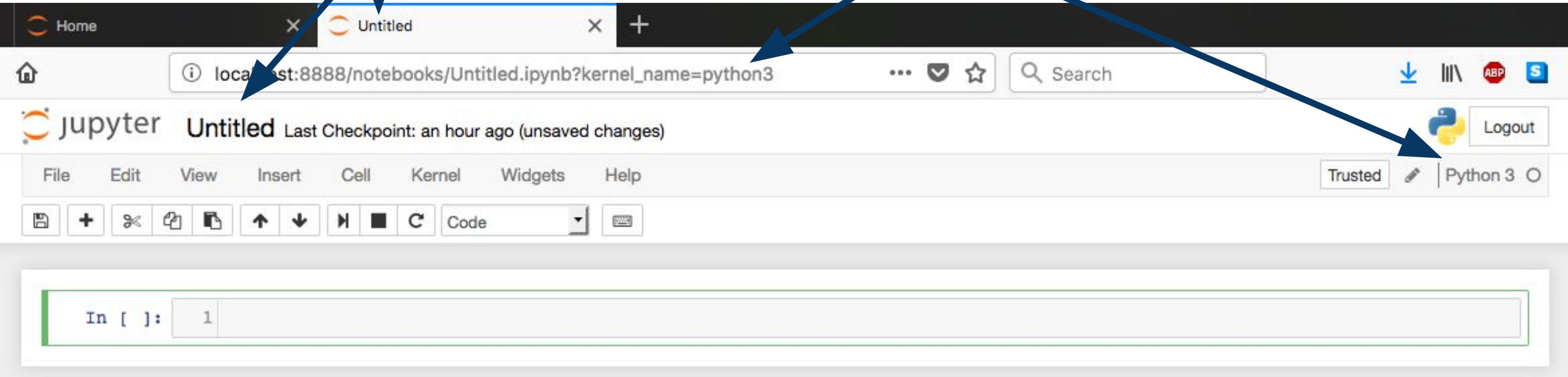
Running Python 3



Home
Untitled
localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
Search
jupyter
Untitled Last Checkpoint: an hour ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Code
In [ ]:

Notebook doesn’t have a title, yet.

Running Python 3



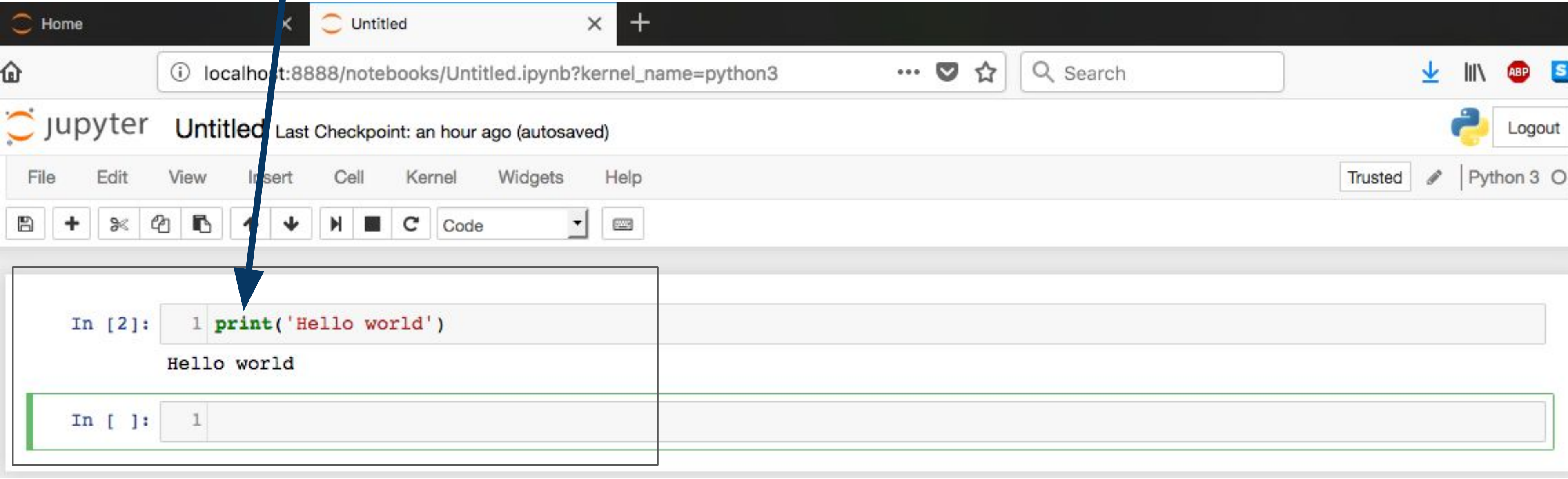
localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3

Untitled Last Checkpoint: an hour ago (unsaved changes)

```
In [ ]:
```

I’ll leave it to you to learn about the other features by reading the documentation. For now, the green-highlighted box is most important. That’s where we write Python code.

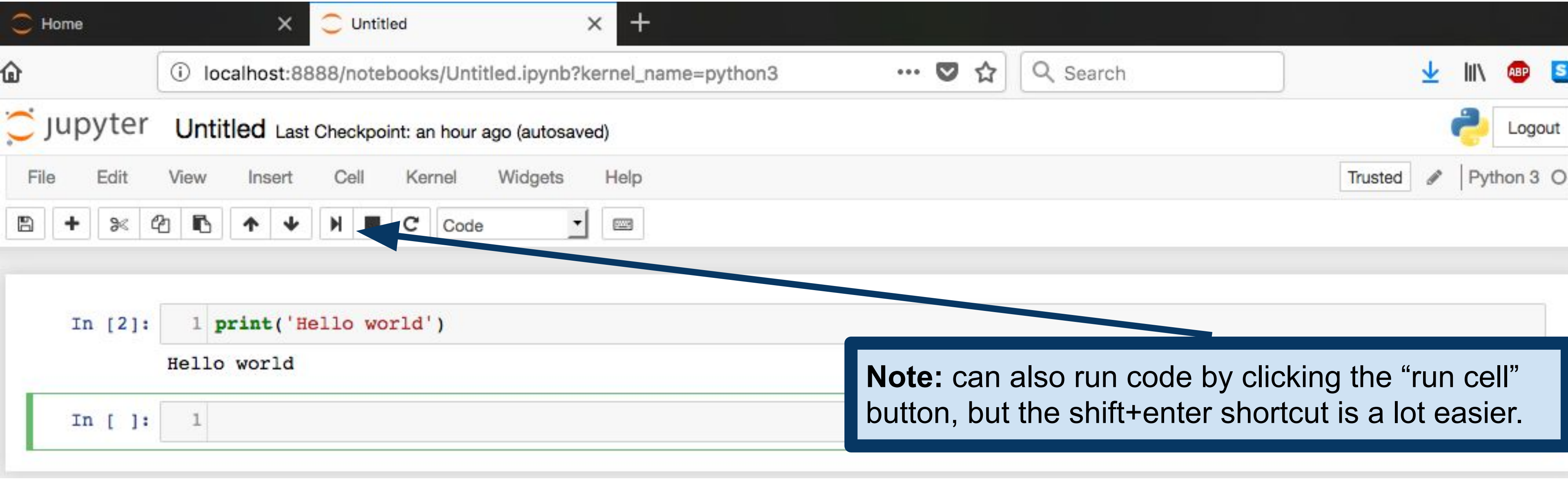
Write code in the highlighted box, then press shift+enter to run the code in that box...



```
print('Hello world')
```

```
Hello world
```

Write code in the highlighted box, then press shift+enter to run the code in that box...

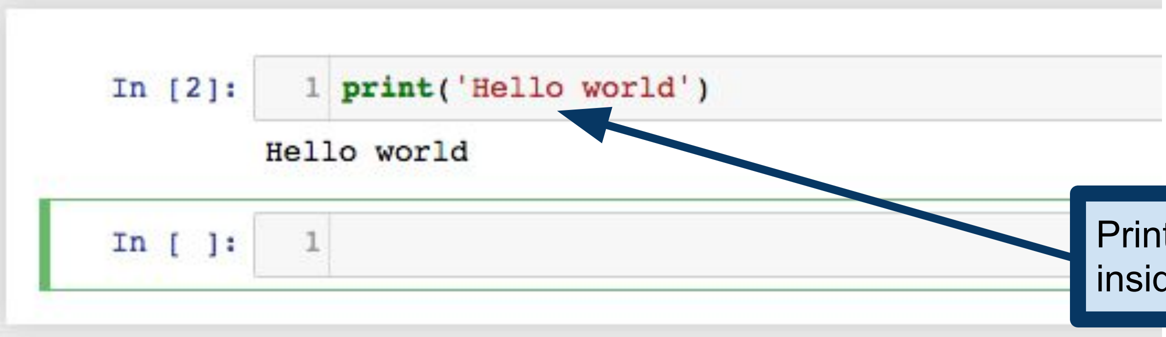

Home
Untitled
localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
Search
jupyter
Untitled Last Checkpoint: an hour ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Code

```
In [2]: print('Hello world')
Hello world

In [ ]:
```

Note: can also run code by clicking the “run cell” button, but the shift+enter shortcut is a lot easier.

Our first function: `print`



```
print('Hello world')
```

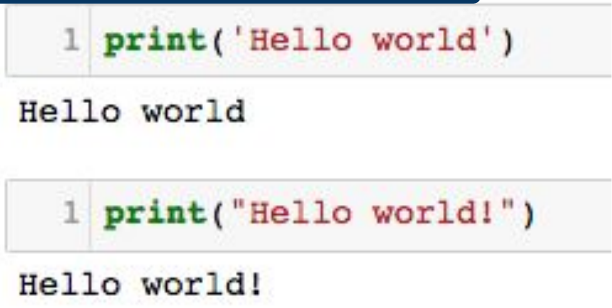
```
Hello world
```

Print displays whatever is inside the quotation marks.

If you haven’t already guessed, print takes a Python **string** and prints it. Of course, “print” here means to display a string, not literally print it on a printer!

Note: if you know Python 2, you’ll notice that print is a bit different in Python 3. That is because in Python 2, print was a **statement**, whereas in Python 3, print is a **function**.

Can also use double quotes



```
print('Hello world')
```

```
Hello world
```

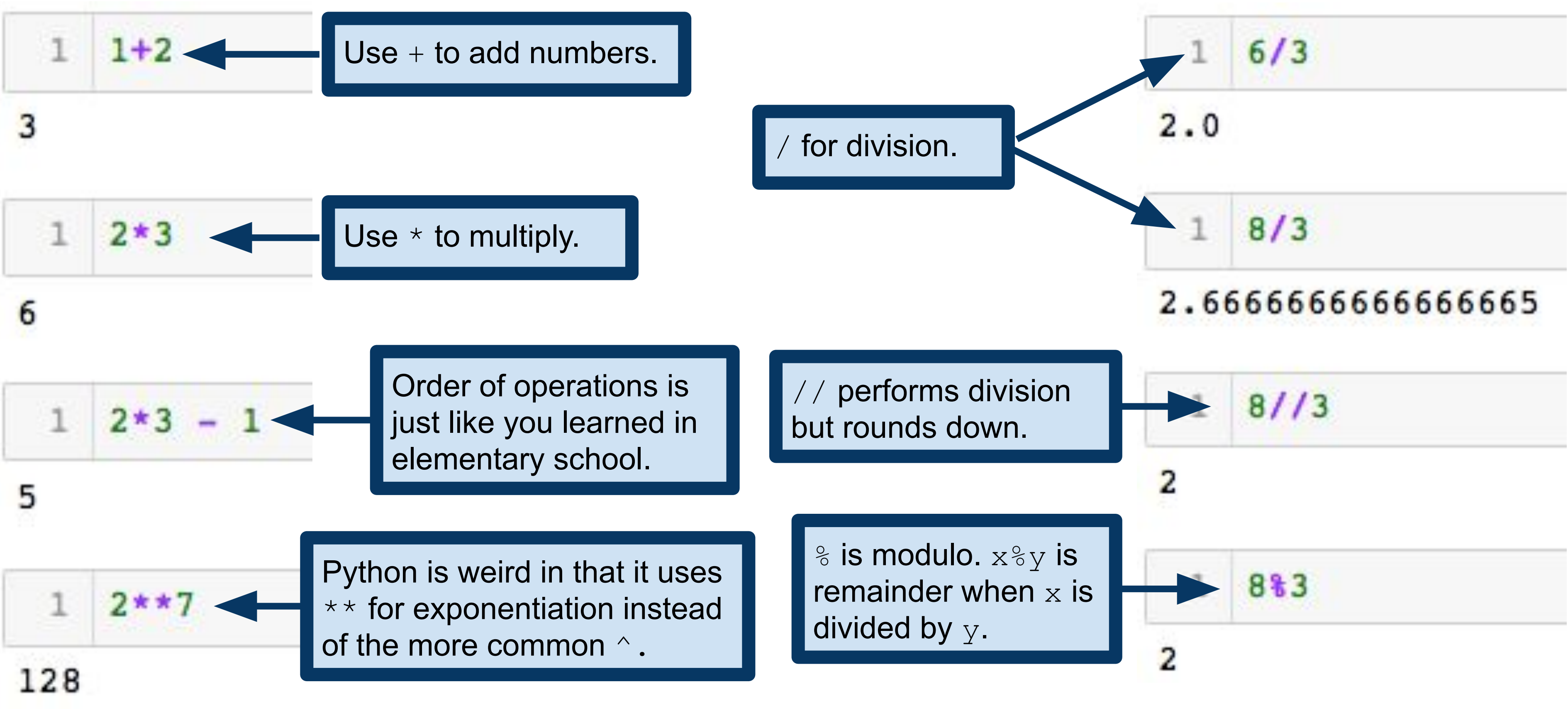
```
print("Hello world!")
```

```
Hello world!
```

Intermission

Lesson 2: Data Types, Functions, and Conditionals

Arithmetic in Python



```
1+2
```

```
3
```

Use + to add numbers.

```
2*3
```

```
6
```

Use * to multiply.

```
2*3 - 1
```

```
5
```

Order of operations is just like you learned in elementary school.

```
2**7
```

```
128
```

Python is weird in that it uses ** for exponentiation instead of the more common ^ .

/ for division.

```
6/3
```

```
2.0
```

```
8/3
```

```
2.6666666666666665
```

// performs division but rounds down.

```
8//3
```

```
2
```

% is modulo. x%y is remainder when x is divided by y.

```
8%3
```

```
2
```

Data Types

Programs work with **values**, which come with different **types**

Examples:

The value `42` is an **integer**

The value `2.71828` is a **floating point number** (i.e., decimal number)

The value `“bird”` is a **string** (i.e., a *string of characters*)

Variable’s type determines what operations we can and can’t perform

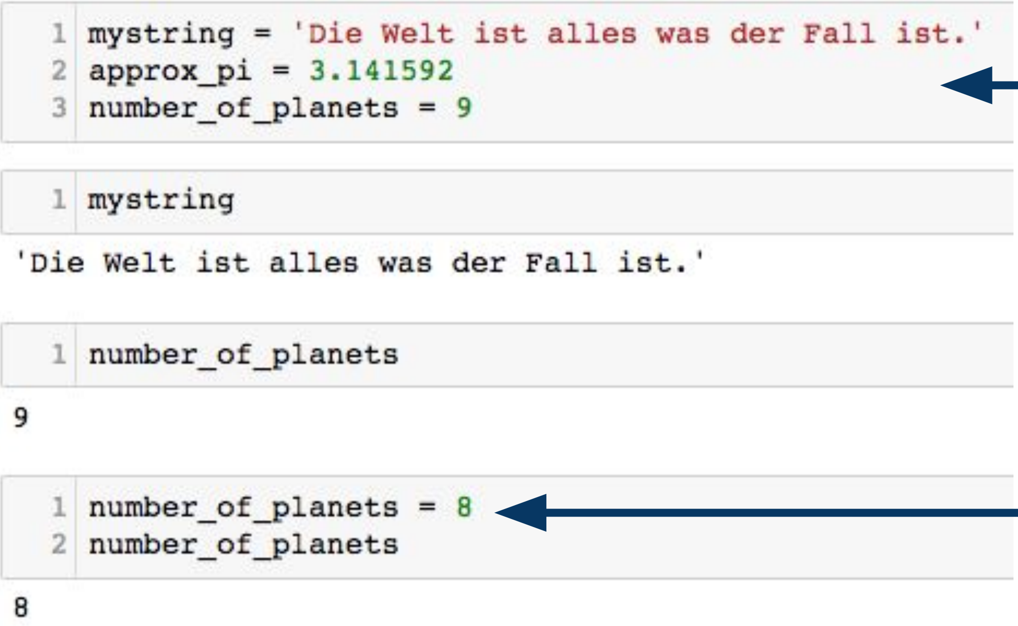
e.g., `2*3` makes sense, but what is `‘cat’*‘dog’`?

(We’ll come back to this in more detail in a few slides)

Variables in Python

Variable is a name that refers to a value

Assign a value to a variable via **variable assignment**



```
mystring = 'Die Welt ist alles was der Fall ist.'
approx_pi = 3.141592
number_of_planets = 9
```

```
mystring
```

```
'Die Welt ist alles was der Fall ist.'
```

```
number_of_planets
```

```
9
```

```
number_of_planets = 8
number_of_planets
```

```
8
```

Assign values to three variables

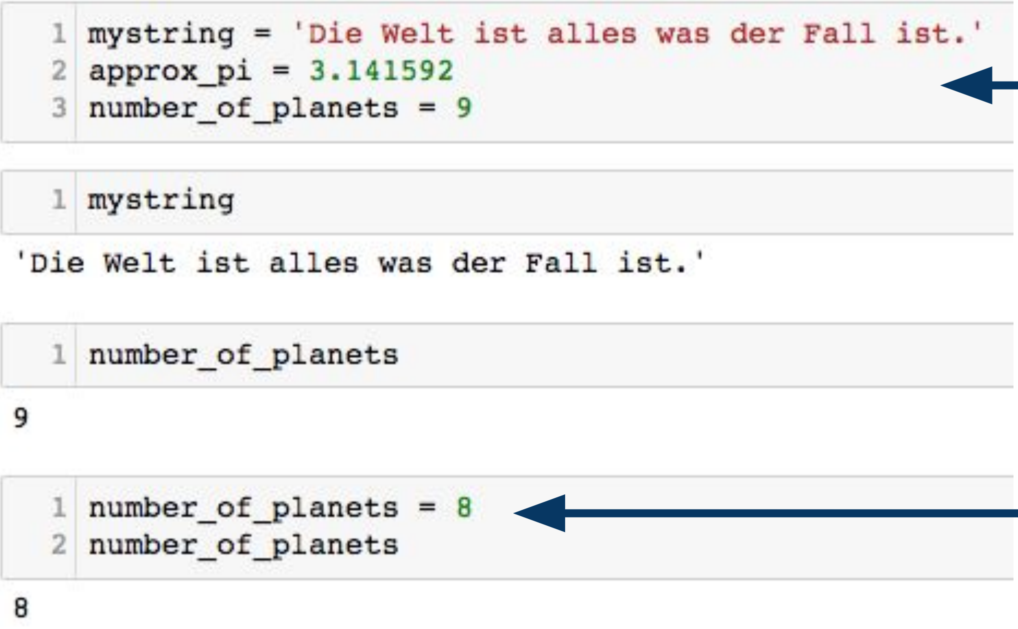
Change the value of `number_of_planets` via another assignment statement.

Variables in Python

Variable is a name that refers to a value

Note: unlike some languages (e.g., C/C++ and Java), you don’t need to tell Python the type of a variable when you declare it. Instead, Python figures out the type of a variable automatically. Python uses what is called **duck typing**, which we will return to in a few lectures.

Assign a value to a variable via **variable assignment**



```
mystring = 'Die Welt ist alles was der Fall ist.'
approx_pi = 3.141592
number_of_planets = 9
```

```
mystring
```

```
'Die Welt ist alles was der Fall ist.'
```

```
number_of_planets
```

```
9
```

```
number_of_planets = 8
number_of_planets
```

```
8
```

Assign values to three variables

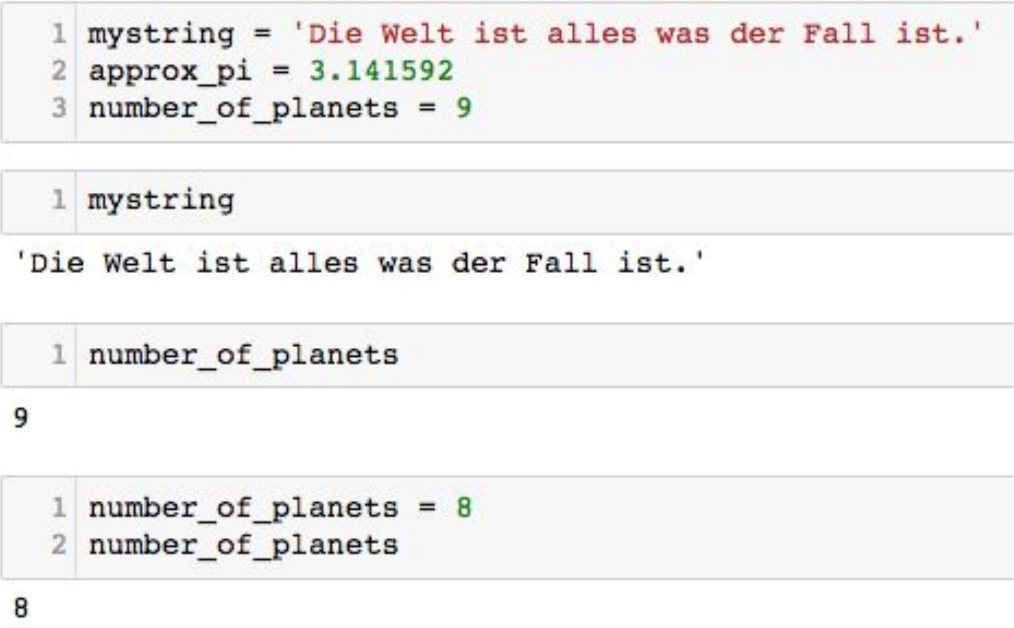
Change the value of `number_of_planets` via another assignment statement.

Variables in Python

Note: unlike some languages (e.g., C/C++ and Java), you don’t need to tell Python the type of a variable when you declare it. Instead, Python figures out the type of a variable automatically. Python uses what is called **duck typing**, which we will return to in a few lectures.

Variable is a name that refers to a value

Assign a value to a variable via **variable assignment**



```
mystring = 'Die Welt ist alles was der Fall ist.'
approx_pi = 3.141592
number_of_planets = 9
```

```
mystring
```

```
'Die Welt ist alles was der Fall ist.'
```

```
number_of_planets
```

```
9
```

```
number_of_planets = 8
number_of_planets
```

```
8
```

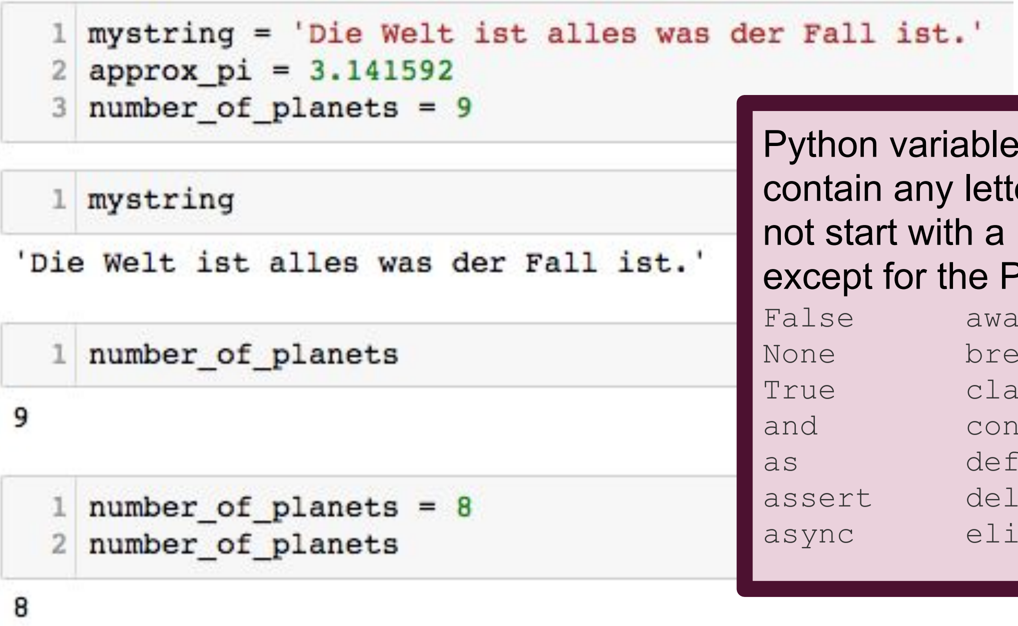
If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck. *https://en.wikipedia.org/wiki/Duck_test*

Variables in Python

Note: unlike some languages (e.g., C/C++ and Java), you don't need to tell Python the type of a variable when you declare it. Instead, Python figures out the type of a variable automatically. Python uses what is called **duck typing**, which we will return to in a few lectures.

Variable is a name that refers to a value

Assign a value to a variable via **variable assignment**



```
mystring = 'Die Welt ist alles was der Fall ist.'
approx_pi = 3.141592
number_of_planets = 9
```

```
mystring
```

```
'Die Welt ist alles was der Fall ist.'
```

```
number_of_planets
```

```
9
```

```
number_of_planets = 8
number_of_planets
```

```
8
```

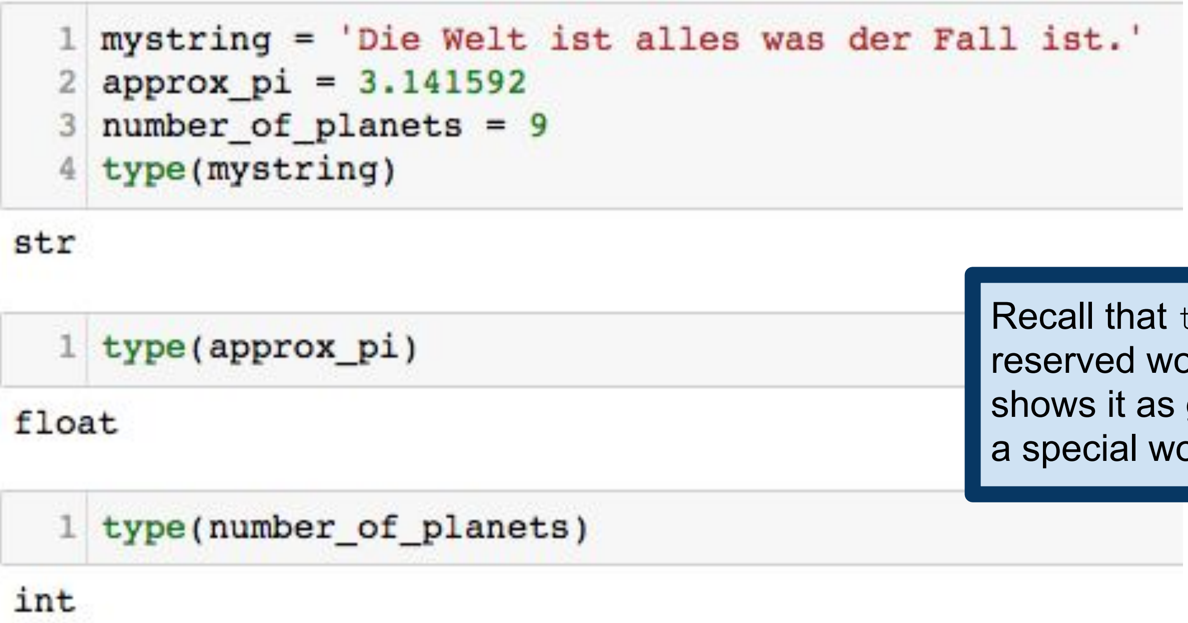
Python variable names can be arbitrarily long, and may contain any letters, numbers and underscore (_), but may not start with a number. Variables can have any name, except for the Python 3 reserved keywords:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Variables in Python

Sometimes we do need to know the type of a variable

Python `type()` function does this for us



```
mystring = 'Die Welt ist alles was der Fall ist.'
approx_pi = 3.141592
number_of_planets = 9
type(mystring)
```

```
str
```

```
type(approx_pi)
```

```
float
```

```
type(number_of_planets)
```

```
int
```

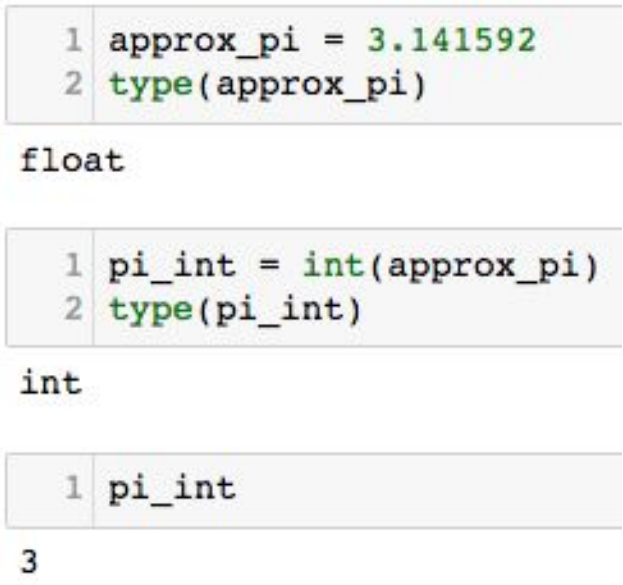
Recall that `type` is one of the Python reserved words. Syntax highlighting shows it as green, indicating that it is a special word in Python.

Variables in Python

Note: changing a variable to a different type is often called **casting** a variable to that type.

We can (sometimes) change the type of a Python variable

Convert a `float` to an `int`:



```
approx_pi = 3.141592
type(approx_pi)
```

```
float
```

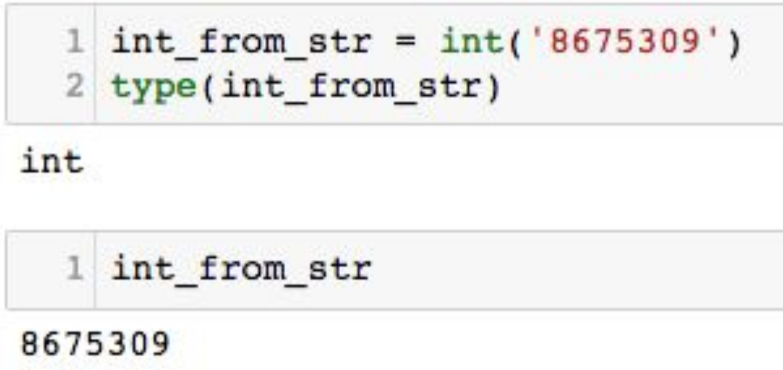
```
pi_int = int(approx_pi)
type(pi_int)
```

```
int
```

```
pi_int
```

```
3
```

Convert a `string` to an `int`:



```
int_from_str = int('8675309')
type(int_from_str)
```

```
int
```

```
int_from_str
```

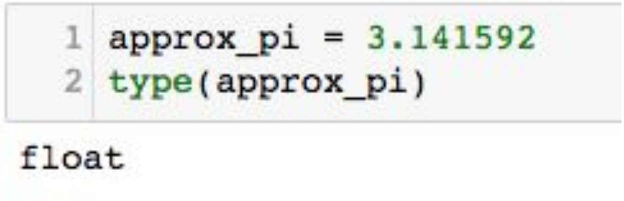
```
8675309
```

Variables in Python

Note: changing a variable to a different type is often called **casting** a variable to that type.

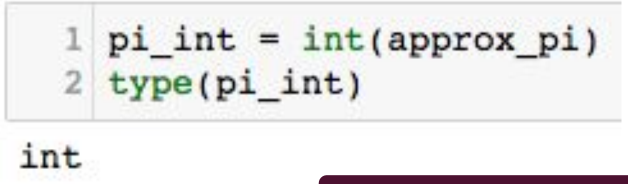
We can (sometimes) change the type of a Python variable

Convert a `float` to an `int`:



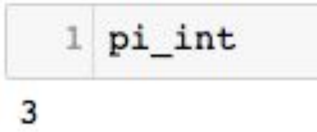
```
approx_pi = 3.141592
type(approx_pi)
```

```
float
```



```
pi_int = int(approx_pi)
type(pi_int)
```

```
int
```

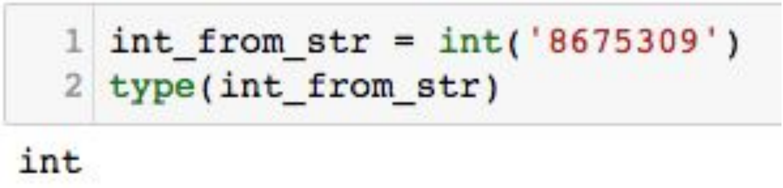


```
pi_int
```

```
3
```

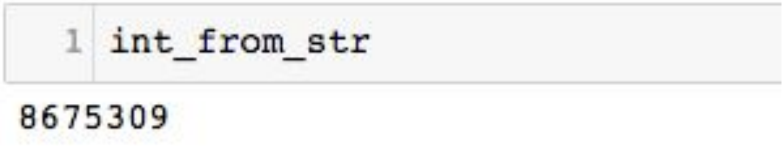
Test your understanding: what should be the value of `float_from_int`?

Convert a `string` to an `int`:



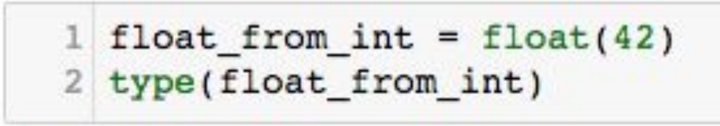
```
int_from_str = int('8675309')
type(int_from_str)
```

```
int
```



```
int_from_str
```

```
8675309
```



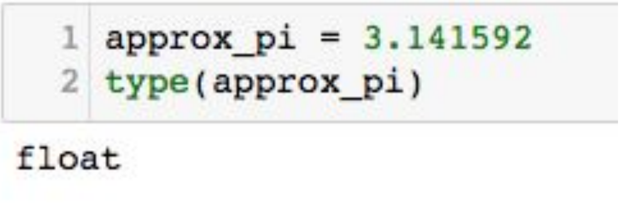
```
float_from_int = float(42)
type(float_from_int)
```

Variables in Python

Note: changing a variable to a different type is often called **casting** a variable to that type.

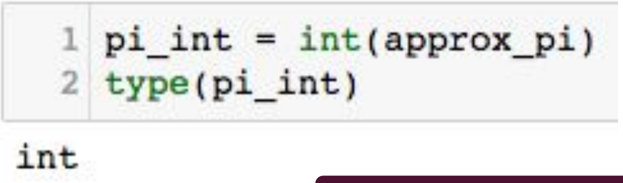
We can (sometimes) change the type of a Python variable

Convert a `float` to an `int`:



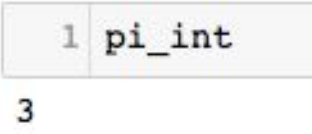
```
approx_pi = 3.141592
type(approx_pi)
```

```
float
```



```
pi_int = int(approx_pi)
type(pi_int)
```

```
int
```

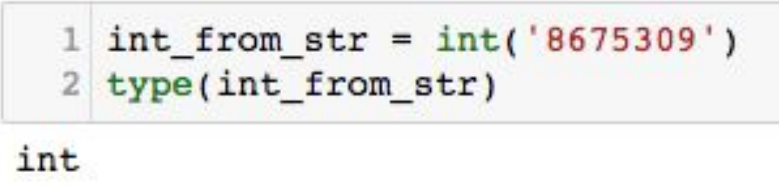


```
pi_int
```

```
3
```

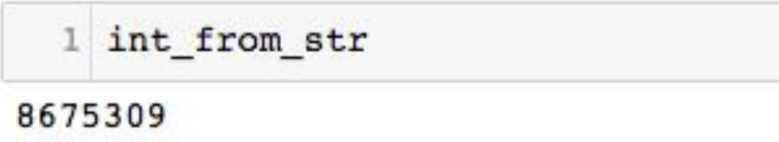
Test your understanding: what should be the value of `float_from_int`?

Convert a `string` to an `int`:



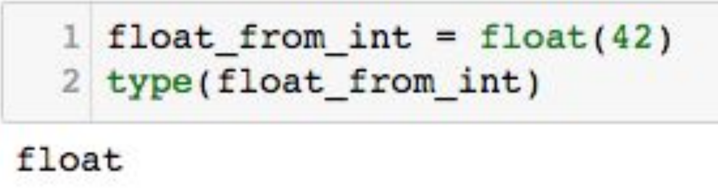
```
int_from_str = int('8675309')
type(int_from_str)
```

```
int
```



```
int_from_str
```

```
8675309
```



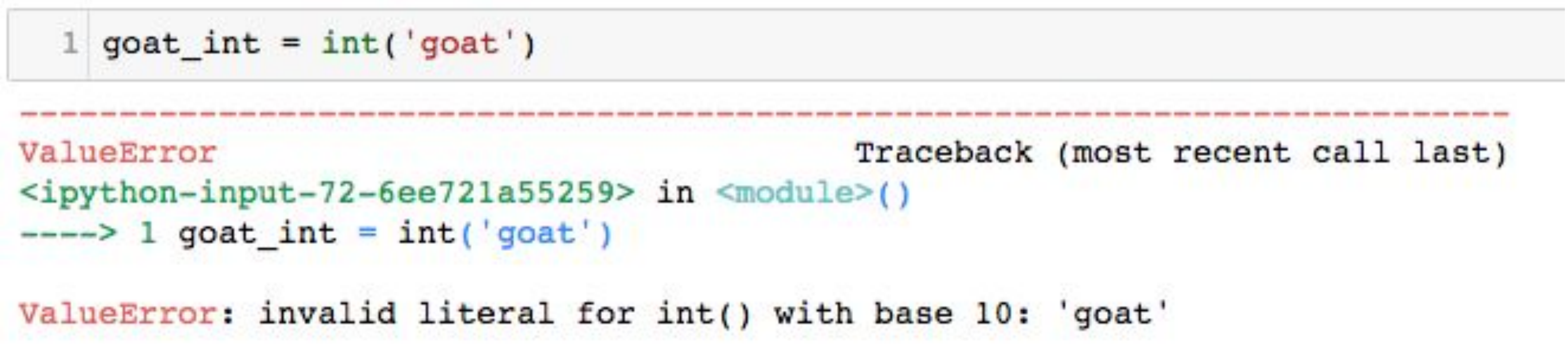
```
float_from_int = float(42)
type(float_from_int)
```

```
float
```

Variables in Python

We can (sometimes) change the type of a Python variable

But if we try to cast to a type that doesn't make sense...



```python
goat_int = int('goat')
```

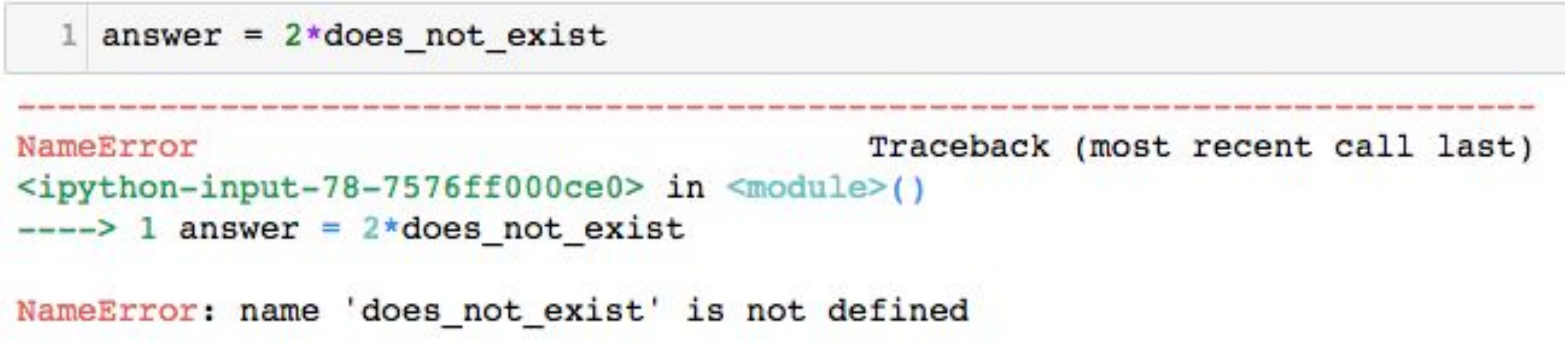
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-72-6ee721a55259> in <module>()
----> 1 goat_int = int('goat')

ValueError: invalid literal for int() with base 10: 'goat'
```

`ValueError` signifies that the type of a variable is okay, but its value doesn't make sense for the operation that we are asking for. https://docs.python.org/3/library/exceptions.html#ValueError

Variables in Python

Variables must be declared (i.e., must have a value) before we evaluate them



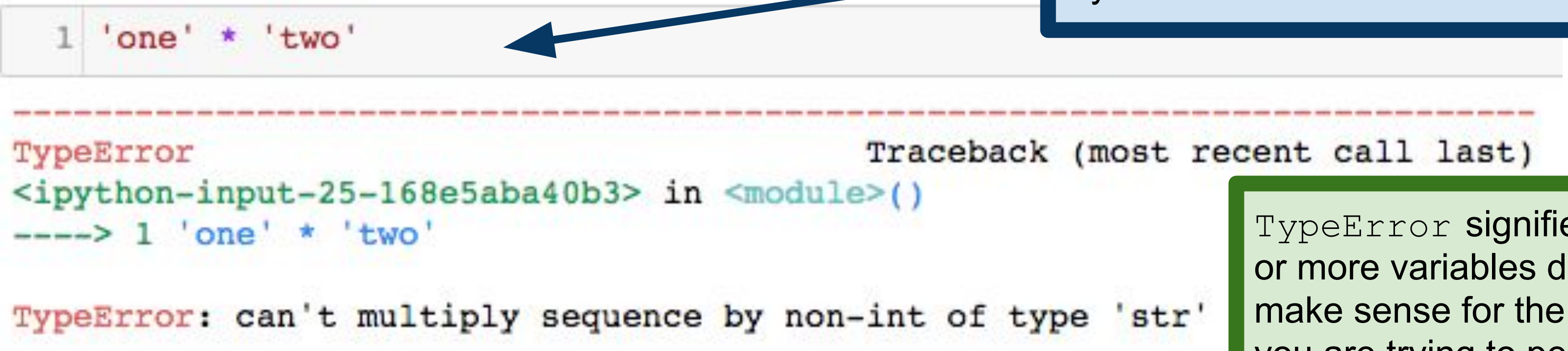
```python
answer = 2*does_not_exist
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-78-7576ff000ce0> in <module>()
----> 1 answer = 2*does_not_exist

NameError: name 'does_not_exist' is not defined
```

`NameError` signifies that Python can’t find anything (variable, function, etc) matching a given name. https://docs.python.org/3/library/exceptions.html#NameError

String Operations



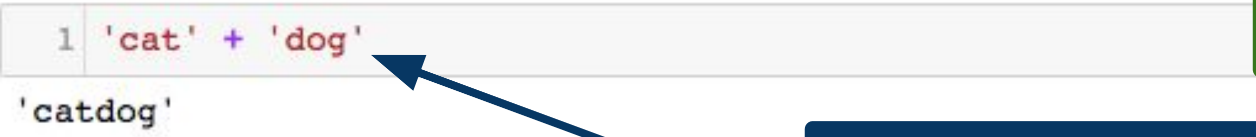
```
'one' * 'two'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-168e5aba40b3> in <module>()
----> 1 'one' * 'two'

TypeError: can't multiply sequence by non-int of type 'str'
```

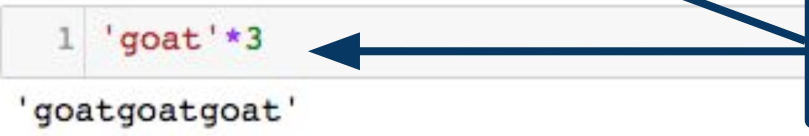
Try to multiply two strings and Python throws an error.

`TypeError` signifies that one or more variables doesn’t make sense for the operation you are trying to perform. https://docs.python.org/3/library/exceptions.html#TypeError



```
'cat' + 'dog'
```

```
'catdog'
```



```
'goat'*3
```

```
'goatgoatgoat'
```

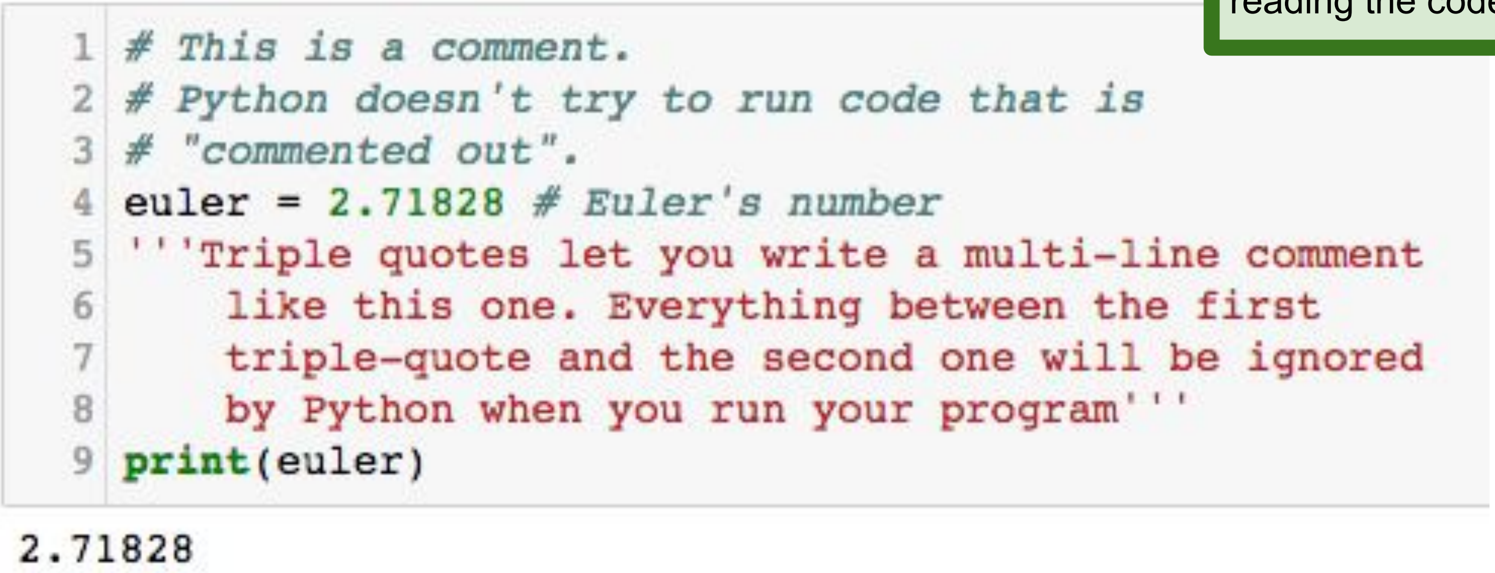
Python uses + to mean **string concatenation**, and defines multiplication of a string by a scalar in the analogous way.

Comments in Python

Comments provide a way to document your code

- Good for when other people have to read your code
- But *also* good for you!

Comments explain to a reader (whether you or someone else) what your code is *meant* to do, which is not always obvious from reading the code itself!



```
# This is a comment.
# Python doesn't try to run code that is
# "commented out".
euler = 2.71828 # Euler's number
'''Triple quotes let you write a multi-line comment
    like this one. Everything between the first
    triple-quote and the second one will be ignored
    by Python when you run your program'''
print(euler)
```

```
2.71828
```

Functions in Python

We’ve already seen examples of functions: e.g., `type()` and `print()`

Function calls take the form `function_name(function arguments)`

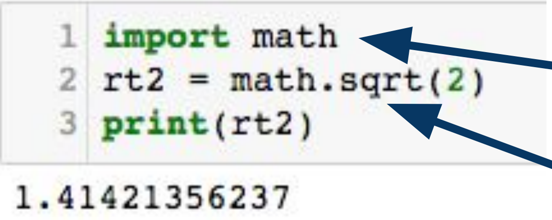
A function takes zero or more **arguments** and **returns** a value

Functions in Python

We’ve already seen examples of functions: e.g., `type()` and `print()`

Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value

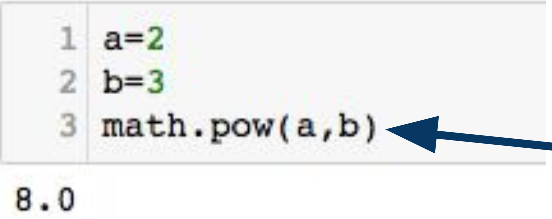


```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```

Python math **module** provides a number of math functions. We have to **import** (i.e., load) the module before we can use it.

`math.sqrt()` takes one argument, returns its square root.



```
a=2
b=3
math.pow(a,b)
```

```
8.0
```

`math.pow()` takes two arguments. Returns the value obtained by raising the first to the power of the second.

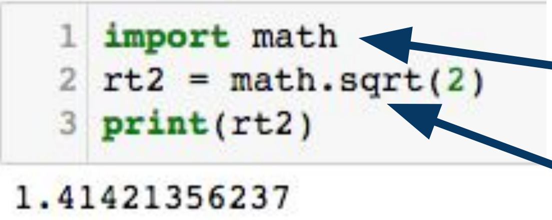
Functions in Python

Note: in the examples below, we write `math.sqrt()` to call the `sqrt()` function from the `math` module. This "dot" notation will show up a lot this semester, so get used to it!

We've already seen examples of functions: e.g., `type()` and `print()`

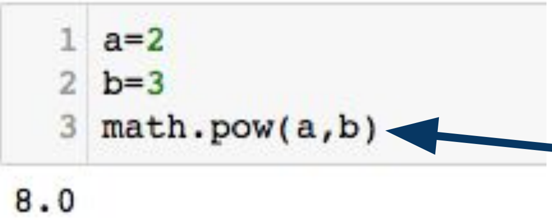
Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value



```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```



```
a=2
b=3
math.pow(a,b)
```

```
8.0
```

Python math **module** provides a number of math functions. We have to **import** (i.e., load) the module before we can use it.

`math.sqrt()` takes one argument, returns its square root.

`math.pow()` takes two arguments. Returns the value obtained by raising the first to the power of the second.

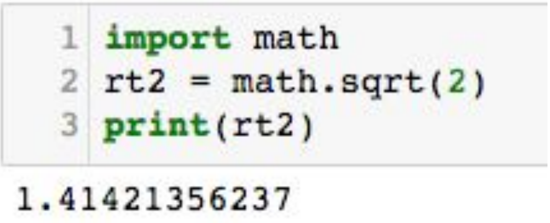
Functions in Python

Note: in the examples below, we write `math.sqrt()` to call the `sqrt()` function from the `math` module. This notation will show up a lot this semester, so get used to it!

We've already seen examples of functions: e.g., `type()` and `print()`

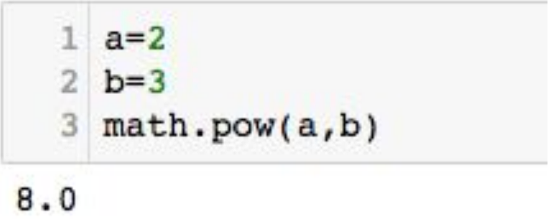
Function calls take the form `function_name(function arguments)`

A function takes zero or more **arguments** and **returns** a value



```
import math
rt2 = math.sqrt(2)
print(rt2)
```

```
1.41421356237
```



```
a=2
b=3
math.pow(a,b)
```

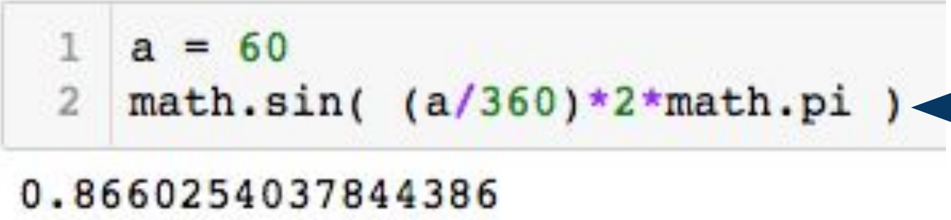
```
8.0
```

Documentation for the Python `math` module: https://docs.python.org/3/library/math.html

Functions in Python

Functions can be **composed**

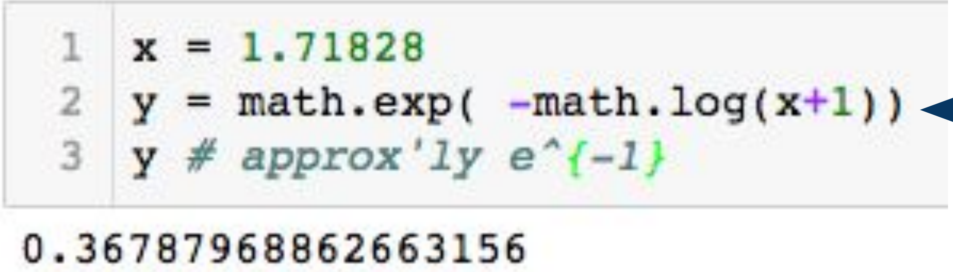
- Supply an expression as the argument of a function
- Output of one function becomes input to another



```
a = 60
math.sin( (a/360)*2*math.pi )
```

```
0.8660254037844386
```

`math.sin()` has as its argument an expression, which has to be evaluated before we can compute the answer.



```
x = 1.71828
y = math.exp( -math.log(x+1))
y # approx'ly e^{-1}
```

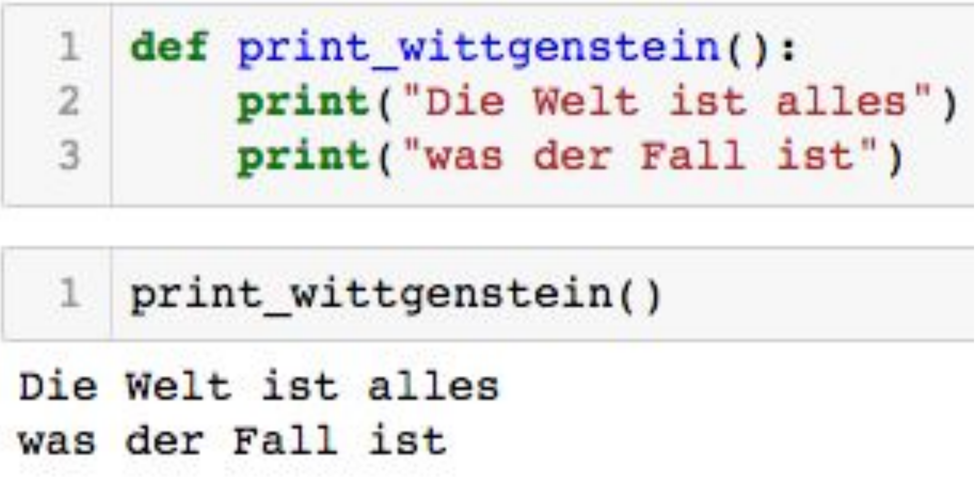
```
0.36787968862663156
```

Functions can even have the outputs of other functions as their arguments.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

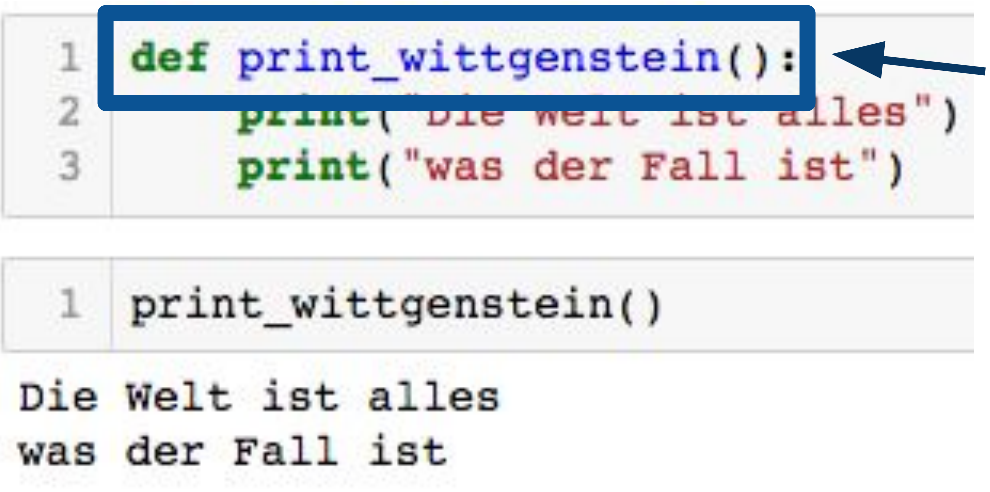
```
Die Welt ist alles
was der Fall ist
```

Let’s walk through this line by line.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```python
print_wittgenstein()
```

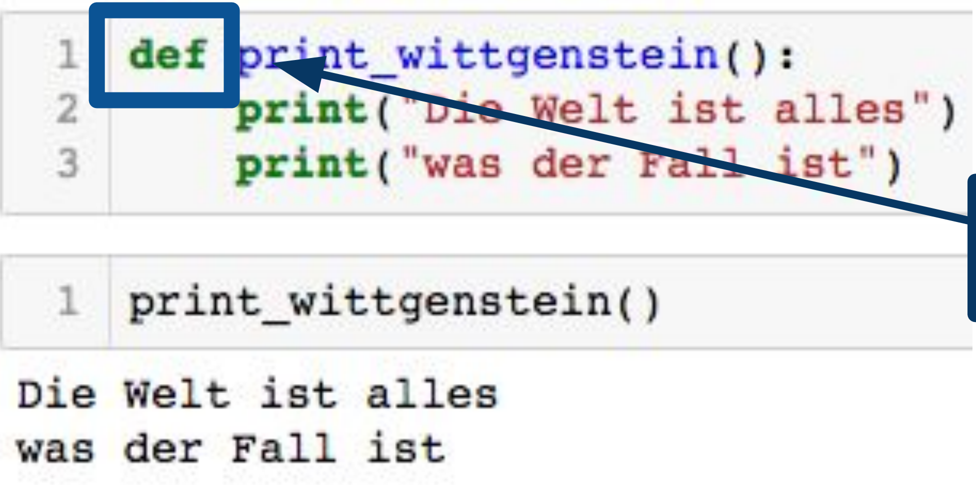
```
Die Welt ist alles
was der Fall ist
```

This line (called the **header** in some documentation) says that we are defining a function called `print_wittgenstein`, and that the function takes no argument.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

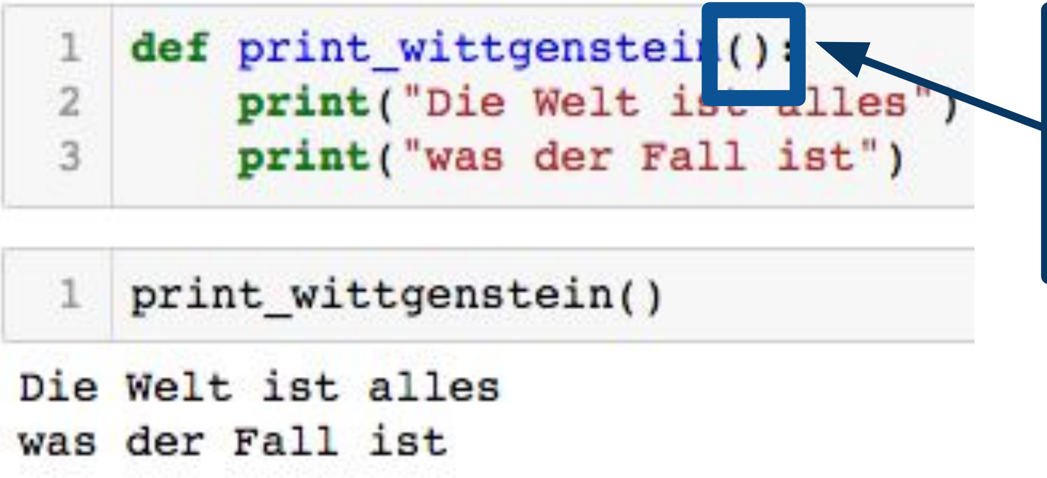
```
Die Welt ist alles
was der Fall ist
```

The `def` keyword tells Python that we are defining a function.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```python
print_wittgenstein()
```

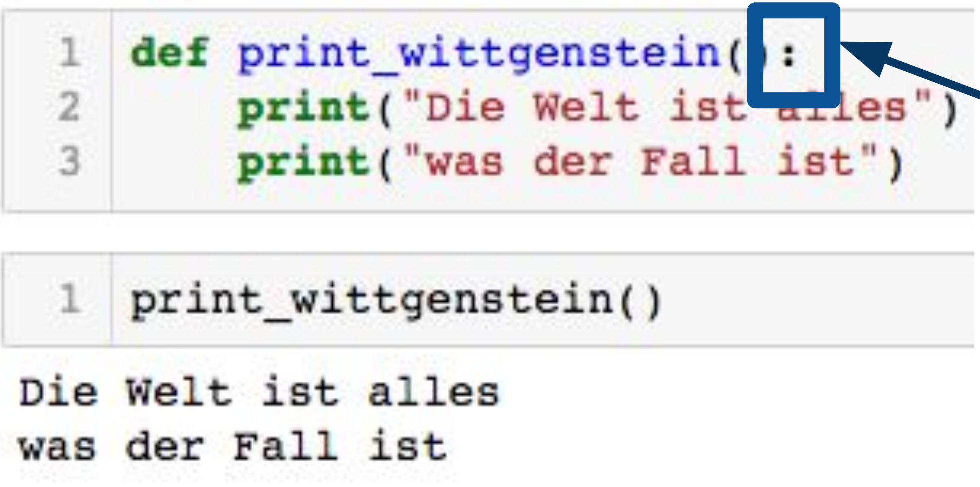
```
Die Welt ist alles
was der Fall ist
```

Any arguments to the function are giving inside the parentheses. This function takes no arguments, so we just give empty parentheses. In a few slides, we’ll see a function that takes arguments.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

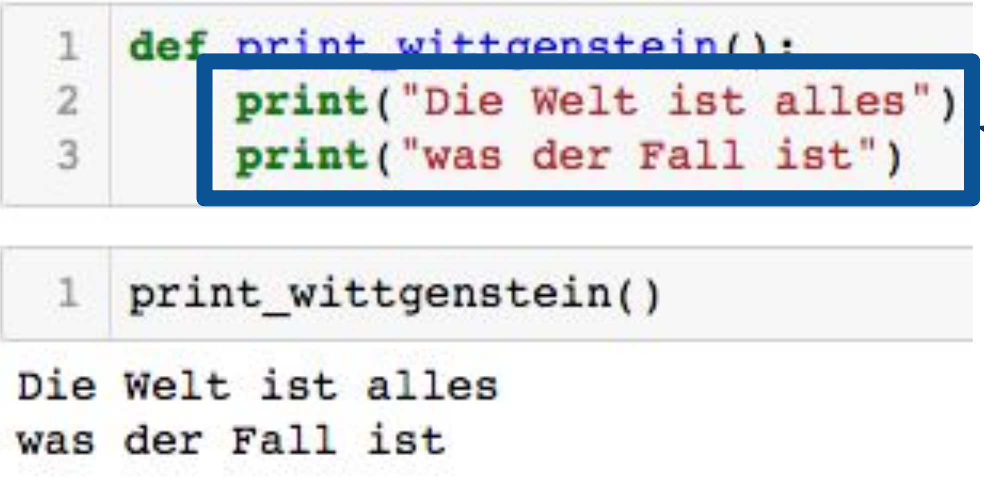
```
Die Welt ist alles
was der Fall ist
```

The colon (`:`) is required by Python’s syntax. You’ll see this symbol a lot, as it is commonly used in Python to signal the start of an indented block of code. (more on this in a few slides).

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

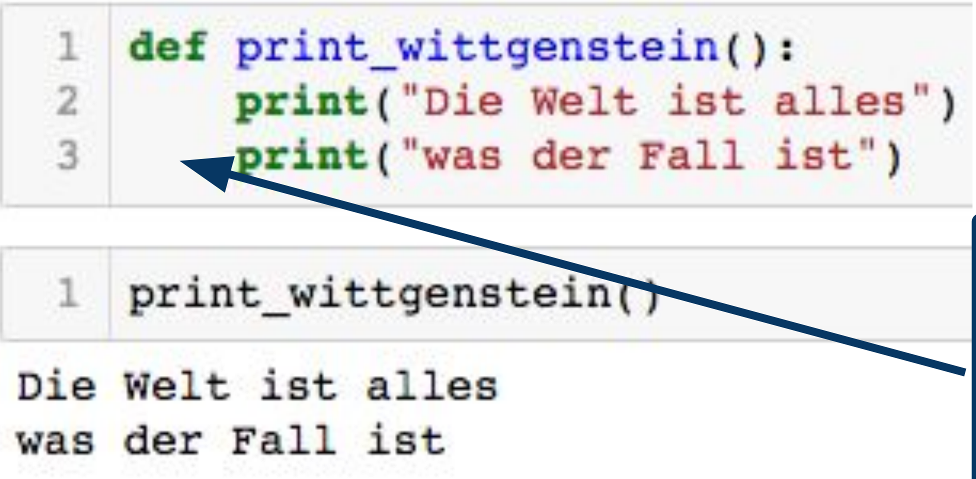
```
Die Welt ist alles
was der Fall ist
```

This is called the **body** of the function. This code is executed whenever the function is called.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```python
print_wittgenstein()
```

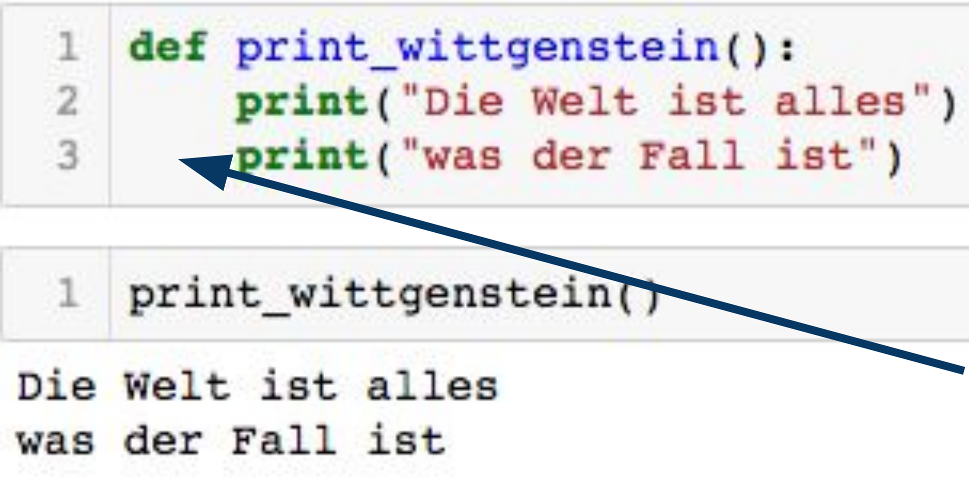
```
Die Welt ist alles
was der Fall ist
```

Note: in languages like R, C/C++ and Java, code is organized into **blocks** using curly braces ({ and }). Python is **whitespace delimited**. So we tell Python which lines of code are part of the function definition using indentation.

Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```

```
print_wittgenstein()
```

```
Die Welt ist alles
was der Fall ist
```

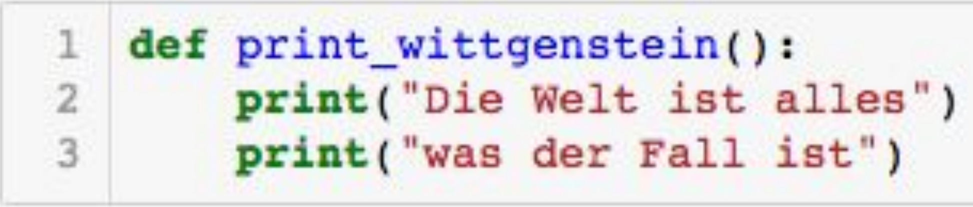
This whitespace can be tabs, or spaces, so long as it’s consistent. It is taken care of automatically by most IDEs.

Note: in languages like R, C/C++ and Java, code is organized into **blocks** using curly braces (`{` and `}`). Python is **whitespace delimited**. So we tell Python which lines of code are part of the function definition using indentation.

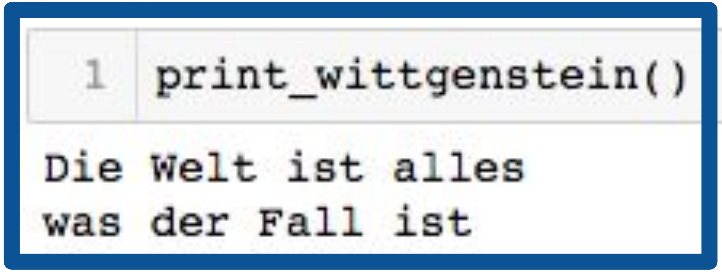
Defining Functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_wittgenstein():
    print("Die Welt ist alles")
    print("was der Fall ist")
```



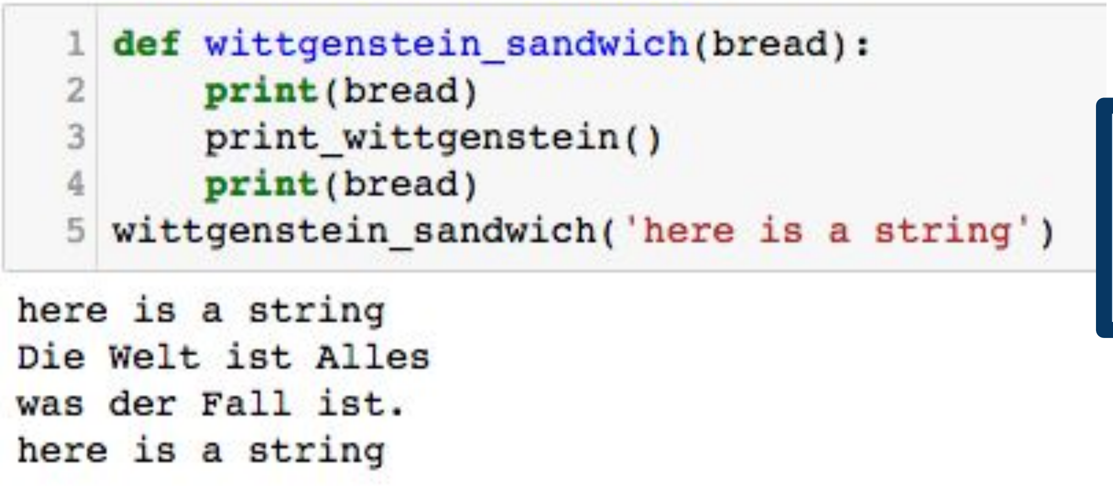
```python
print_wittgenstein()
```

```
Die Welt ist alles
was der Fall ist
```

We have defined our function. Now, any time we call it, Python executes the code in the definition, in order.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



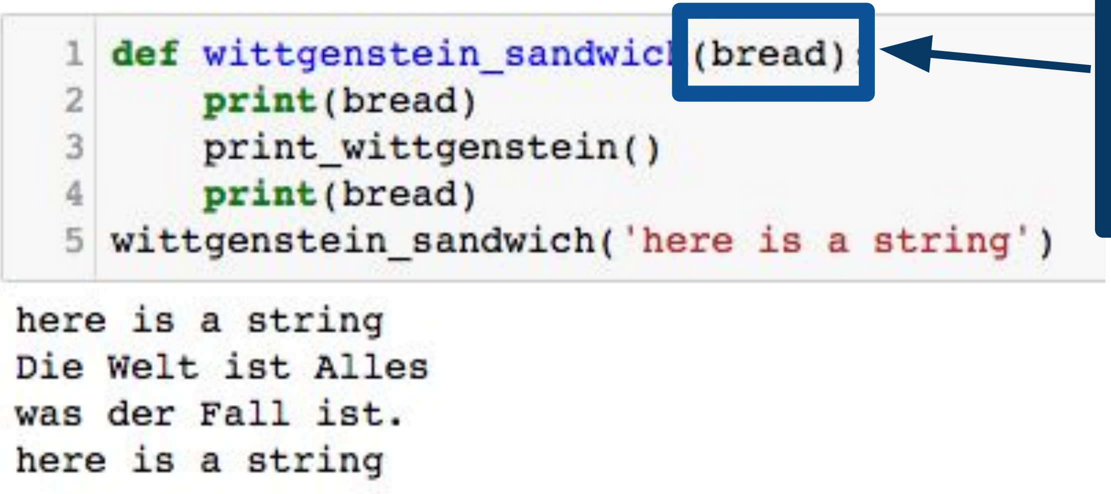
```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

This function takes one argument, prints it, then prints our Wittgenstein quote, then prints the argument again.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



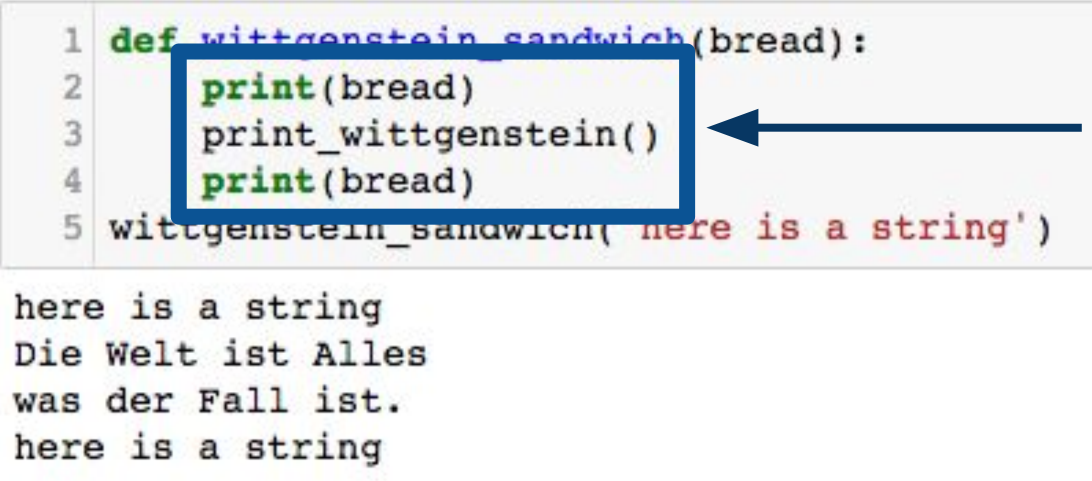
```python
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

This function takes one argument, which we call `bread`. All the arguments named here act like variables **within the body of the function**, but not outside the body. We'll return to this in a few slides.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



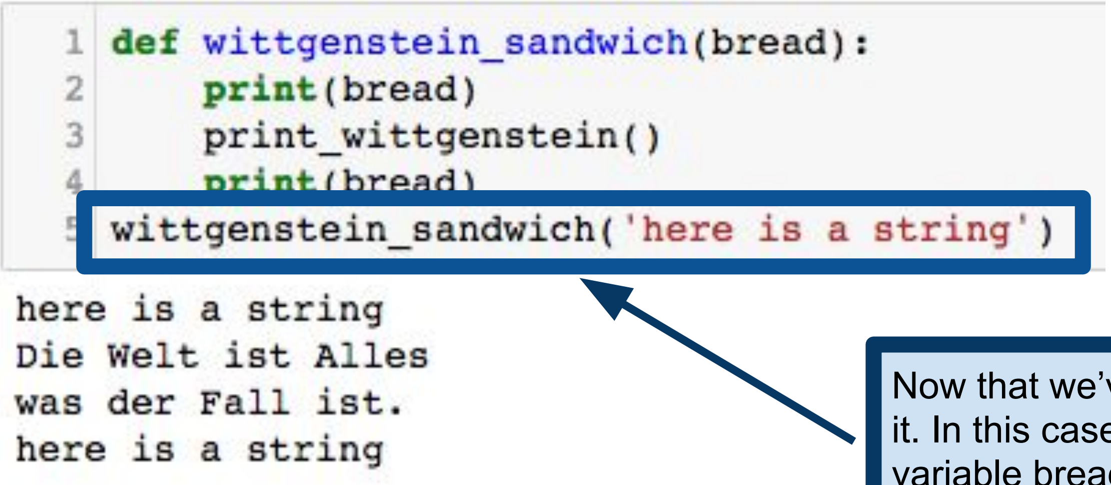
```
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Body of the function specifies what to do with the argument(s). In this case, we print whatever the argument was, then print our Wittgenstein quote, and then print the argument again.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



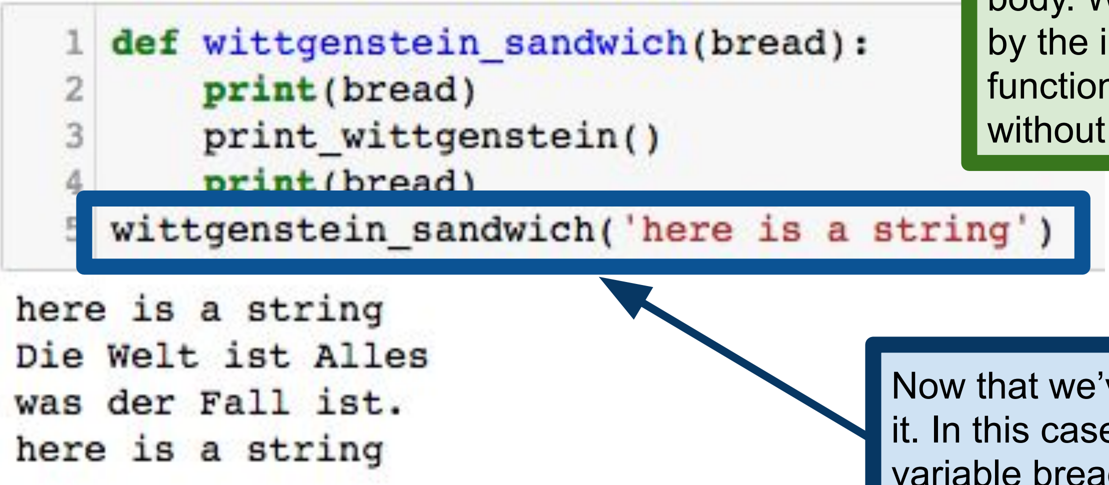
```
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Now that we’ve defined our function, we can call it. In this case, when we call our function, the variable bread in the definition gets the value `‘here is a string’` , and then proceeds to run the code in the function body.

Defining Functions

After defining a function, we can use it anywhere, including in other functions



```
def wittgenstein_sandwich(bread):
    print(bread)
    print_wittgenstein()
    print(bread)
wittgenstein_sandwich('here is a string')
```

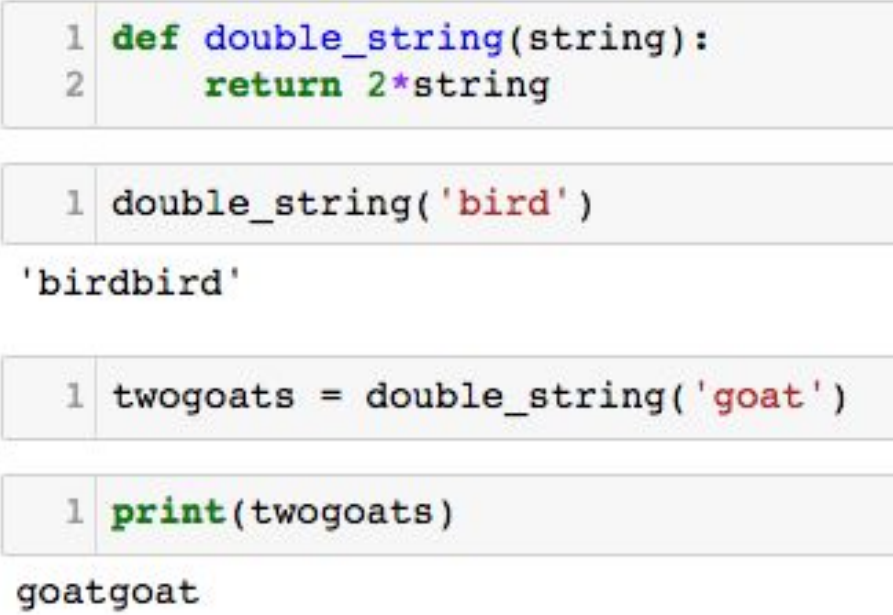
```
here is a string
Die Welt ist Alles
was der Fall ist.
here is a string
```

Note: this last line is **not** part of the function body. We communicate this fact to Python by the indentation. Python knows that the function body is finished once it sees a line without indentation.

Now that we’ve defined our function, we can call it. In this case, when we call our function, the variable bread in the definition gets the value `‘here is a string’`, and then proceeds to run the code in the function body.

Defining Functions

Using the `return` keyword, we can define functions that produce results



```
def double_string(string):
    return 2*string
```

```
double_string('bird')
```

```
'birdbird'
```

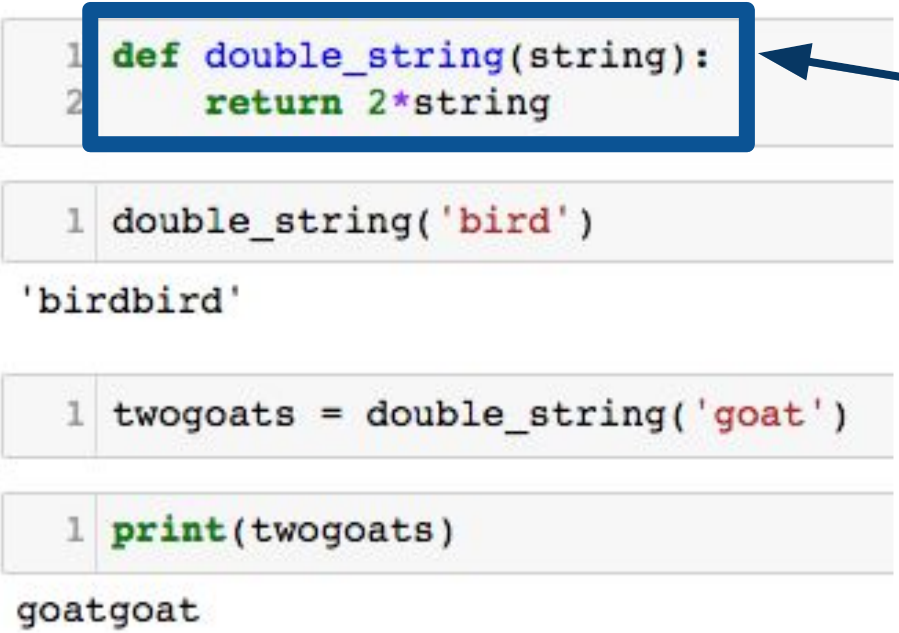
```
twogoats = double_string('goat')
```

```
print(twogoats)
```

```
goatgoat
```

Defining Functions

Using the `return` keyword, we can define functions that produce results



```
def double_string(string):
    return 2*string
```

```
double_string('bird')
```

```
'birdbird'
```

```
twogoats = double_string('goat')
```

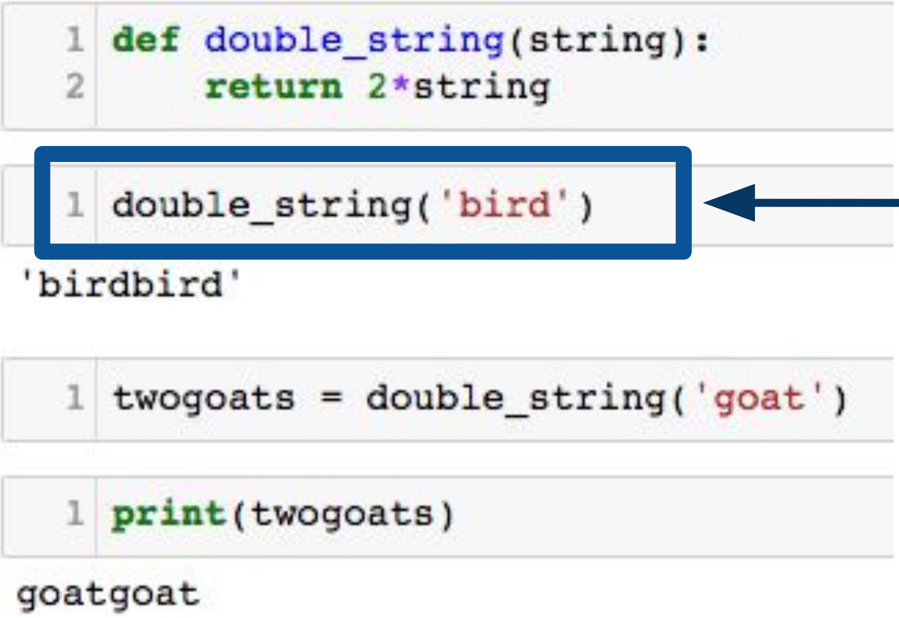
```
print(twogoats)
```

```
goatgoat
```

`double_string` takes one argument, a string, and **returns** that string, concatenated with itself.

Defining Functions

Using the `return` keyword, we can define functions that produce results



```
def double_string(string):
    return 2*string
```

```
double_string('bird')
```

```
'birdbird'
```

```
twogoats = double_string('goat')
```

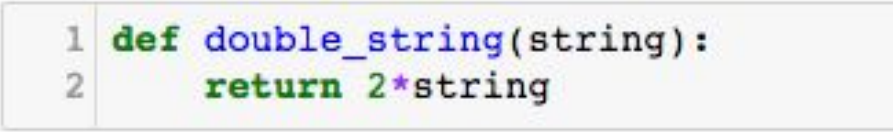
```
print(twogoats)
```

```
goatgoat
```

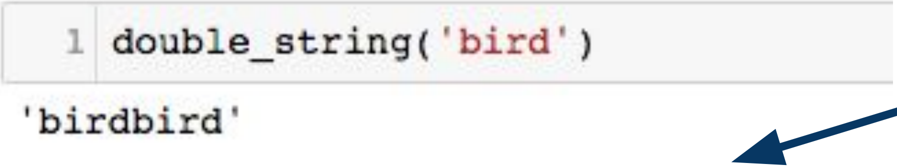
So when Python executes this line, it takes the string `'bird'`, which becomes the parameter `string` in the function `double_string`, and this line **evaluates** to the string `'birdbird'`.

Defining Functions

Using the `return` keyword, we can define functions that produce results

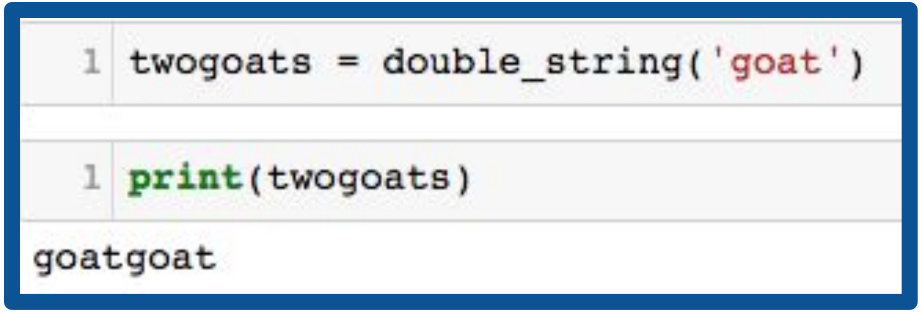


```
def double_string(string):
    return 2*string
```



```
double_string('bird')
```

```
'birdbird'
```



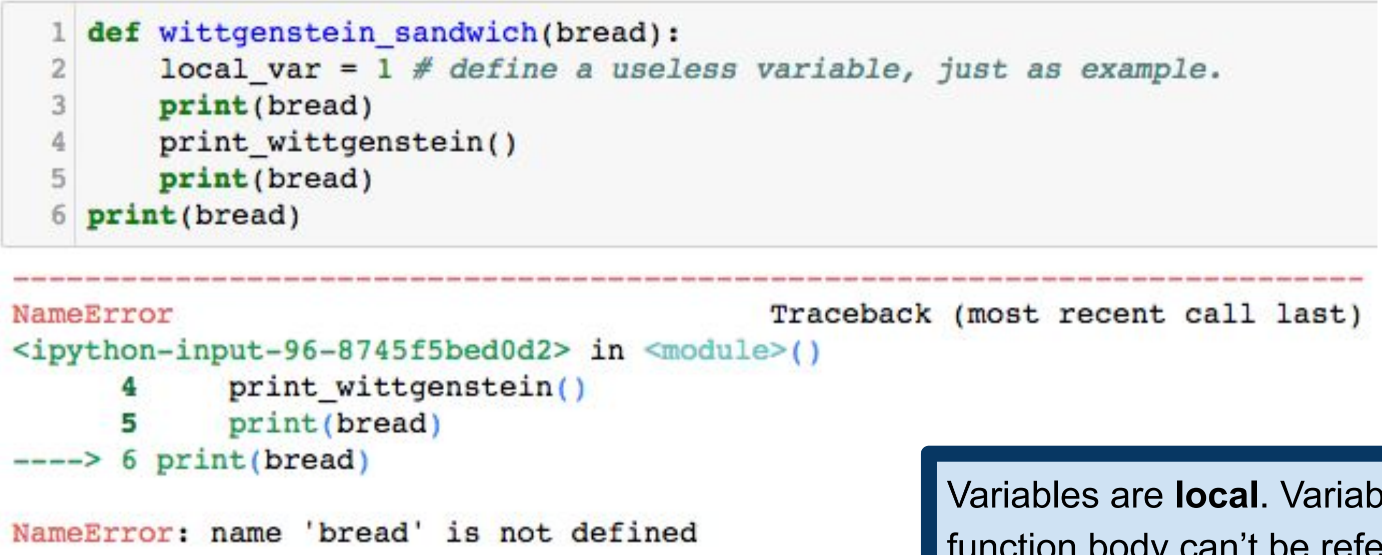
```
twogoats = double_string('goat')
```

```
print(twogoats)
```

```
goatgoat
```

Alternatively, we can call the function and assign its result to a variable, just like we did with the functions in the `math` module.

Defining Functions

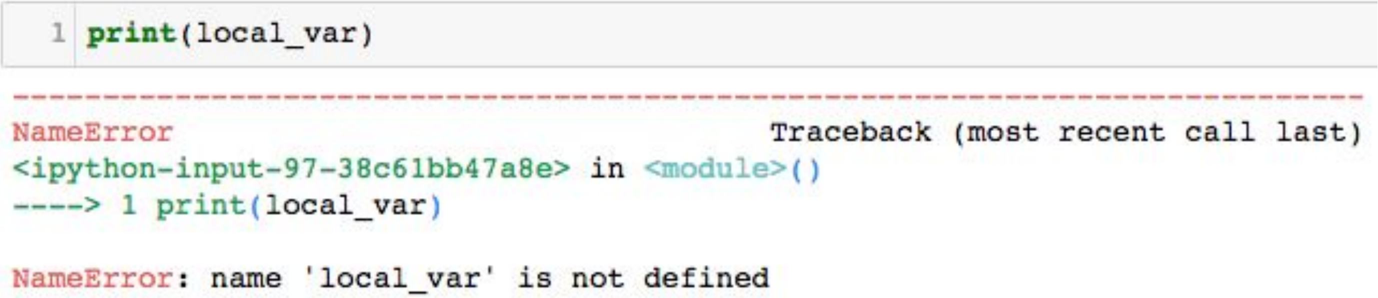


```
def wittgenstein_sandwich(bread):
    local_var = 1 # define a useless variable, just as example.
    print(bread)
    print_wittgenstein()
    print(bread)
print(bread)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-96-8745f5bed0d2> in <module>()
      4     print_wittgenstein()
      5     print(bread)
----> 6 print(bread)

NameError: name 'bread' is not defined
```

Variables are **local**. Variables defined inside a function body can’t be referenced outside.



```
print(local_var)
```

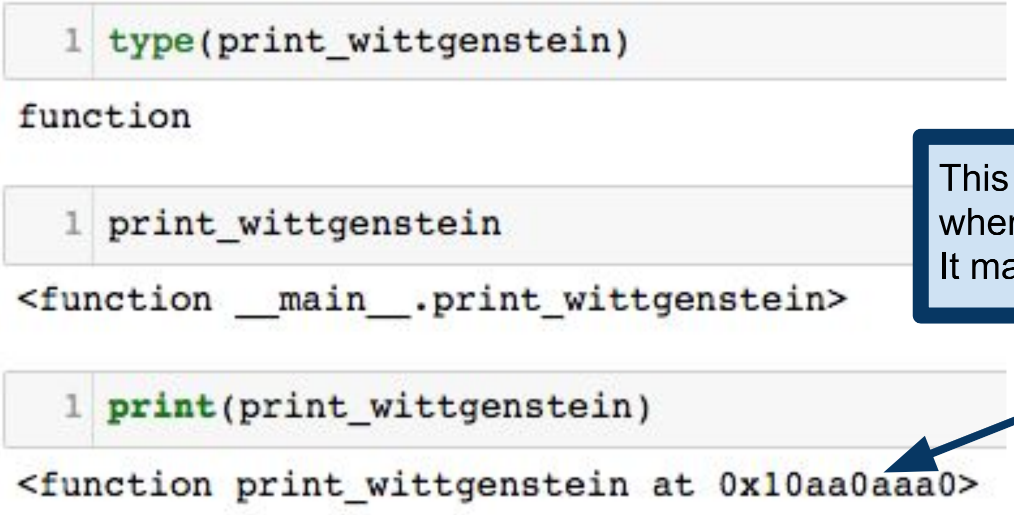
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-97-38c61bb47a8e> in <module>()
----> 1 print(local_var)

NameError: name 'local_var' is not defined
```

Defining Functions

When you define a function, you are actually creating a variable of type **function**

Functions are objects that you can treat just like other variables



```
type(print_wittgenstein)
```

```
function
```

```
print_wittgenstein
```

```
<function __main__.print_wittgenstein>
```

```
print(print_wittgenstein)
```

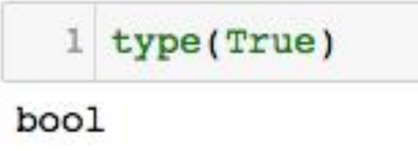
```
<function print_wittgenstein at 0x10aa0aaa0>
```

This number is the address in memory where `print_wittgenstein` is stored. It may be different on your computer.

Boolean Expressions

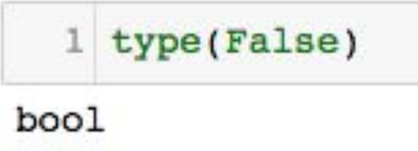
Boolean expressions evaluate the truth/falsity of a statement

Python supplies a special Boolean type, `bool`
variable of type `bool` can be either `True` or `False`



```
type(True)
```

```
bool
```

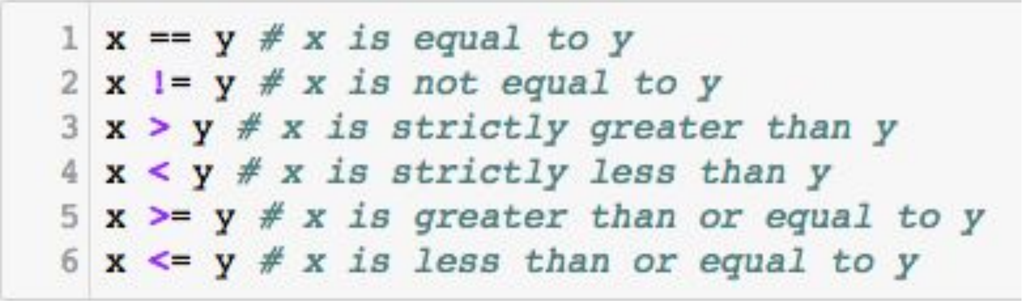


```
type(False)
```

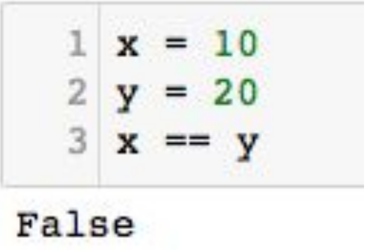
```
bool
```

Boolean Expressions

Comparison operators available in Python:

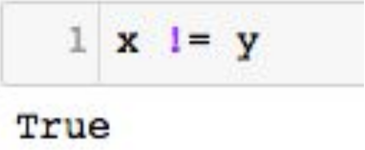


```python
x == y # x is equal to y
x != y # x is not equal to y
x > y # x is strictly greater than y
x < y # x is strictly less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```



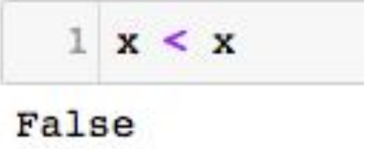
```python
x = 10
y = 20
x == y
```

```
False
```



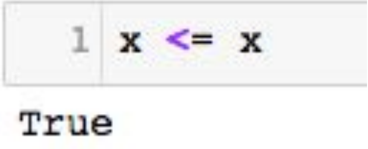
```python
x != y
```

```
True
```



```python
x < x
```

```
False
```



```python
x <= x
```

```
True
```

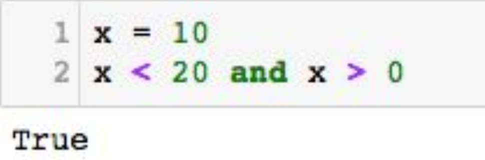
Expressions involving comparison operators evaluate to a Boolean.

Note: In true Pythonic style, one can compare many types, not just numbers. Most obviously, strings can be compared, with ordering given alphabetically.

Boolean Expressions

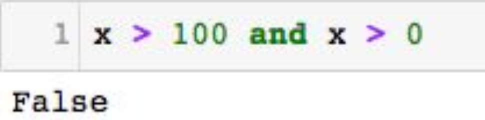
Can combine Boolean expressions into larger expressions via **logical operators**

In Python: `and`, `or` and `not`



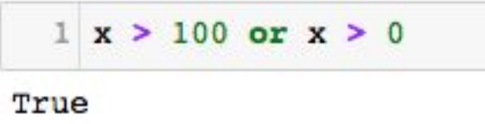
```
x = 10
x < 20 and x > 0
```

```
True
```



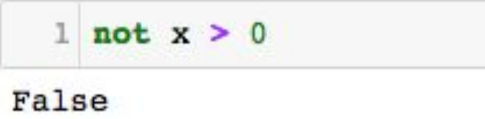
```
x > 100 and x > 0
```

```
False
```



```
x > 100 or x > 0
```

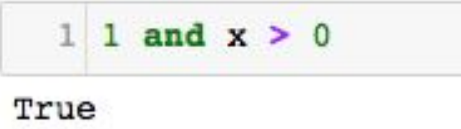
```
True
```



```
not x > 0
```

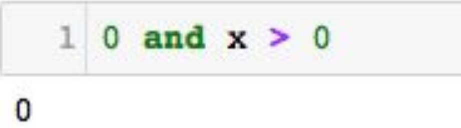
```
False
```

Note: technically, any nonzero number or any nonempty string will evaluate to `True`, but you should avoid comparing anything that isn't Boolean.



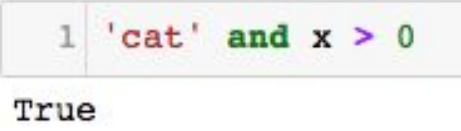
```
1 and x > 0
```

```
True
```



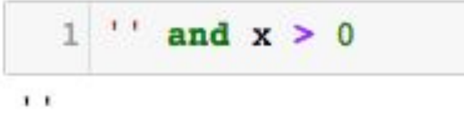
```
0 and x > 0
```

```
0
```



```
'cat' and x > 0
```

```
True
```

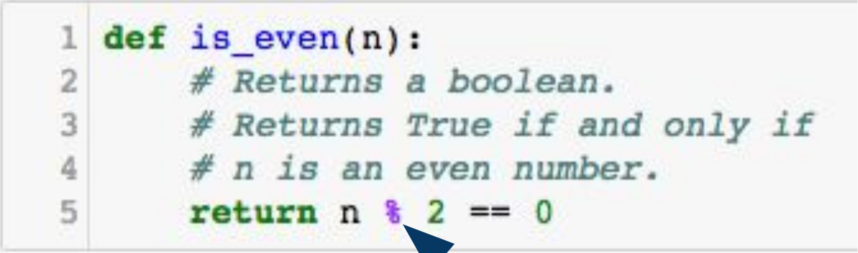


```
'' and x > 0
```

```
''
```

Boolean Expressions: Example

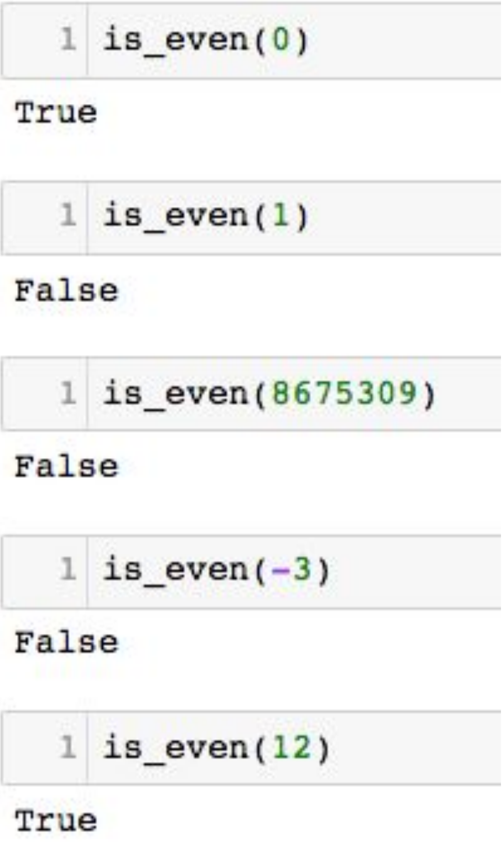
Let’s see Boolean expressions in action



```python
def is_even(n):
    # Returns a boolean.
    # Returns True if and only if
    # n is an even number.
    return n % 2 == 0
```

Reminder: `x % y` returns the remainder when `x` is divided by `y`.

Note: in practice, we would want to include some extra code to check that `n` is actually a number, and to “fail gracefully” if it isn’t, e.g., by throwing an error with a useful error message. More about this in future lectures.



```
is_even(0)
```

```
True
```

```
is_even(1)
```

```
False
```

```
is_even(8675309)
```

```
False
```

```
is_even(-3)
```

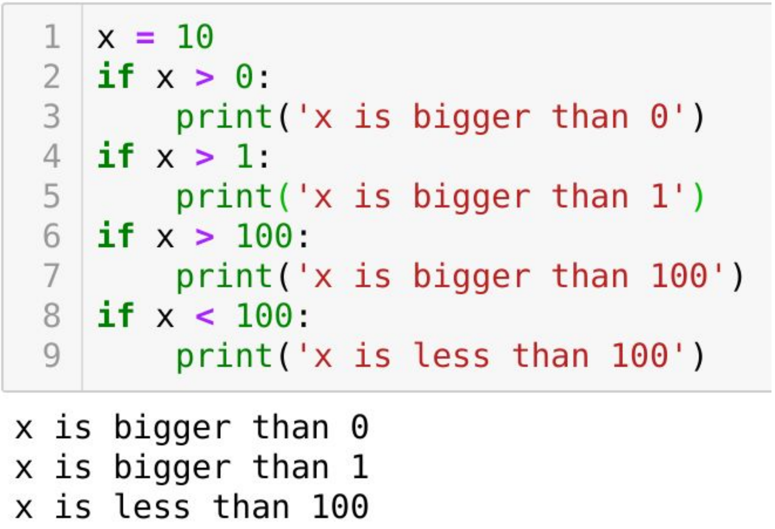
```
False
```

```
is_even(12)
```

```
True
```

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

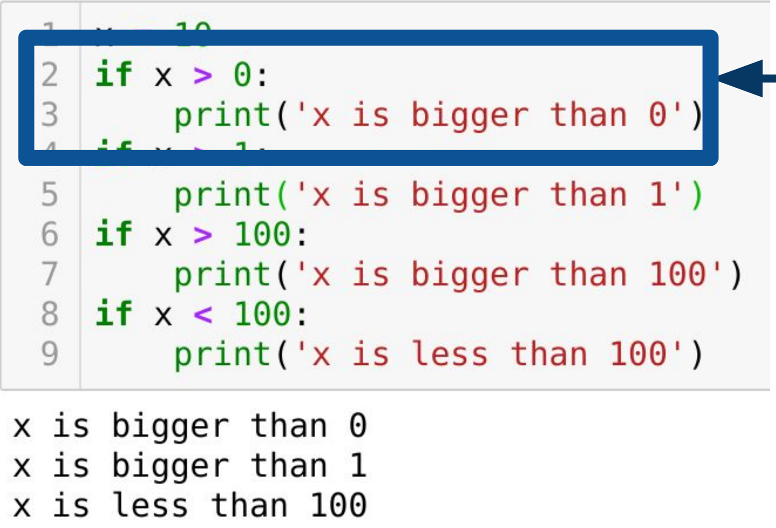


```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



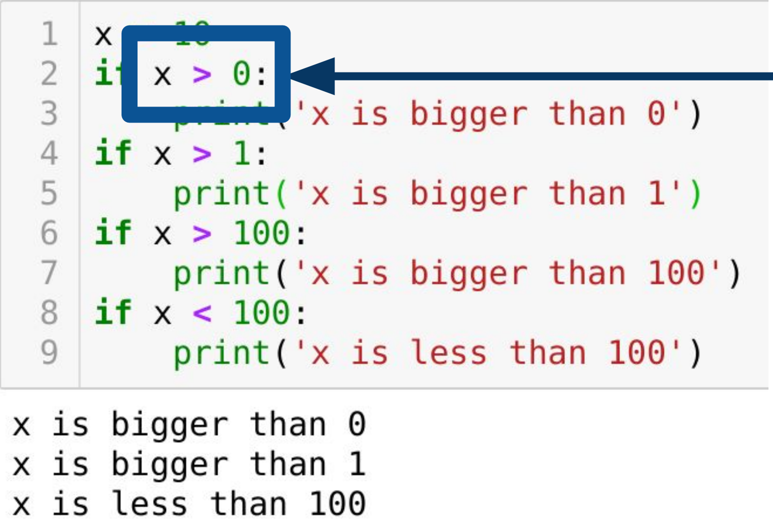
```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This is an **if-statement**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



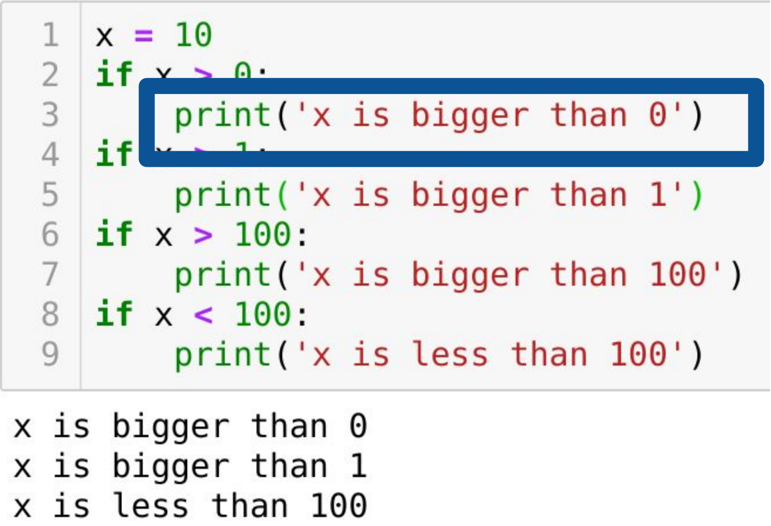
```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This Boolean expression is called the **test condition**, or just the **condition**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



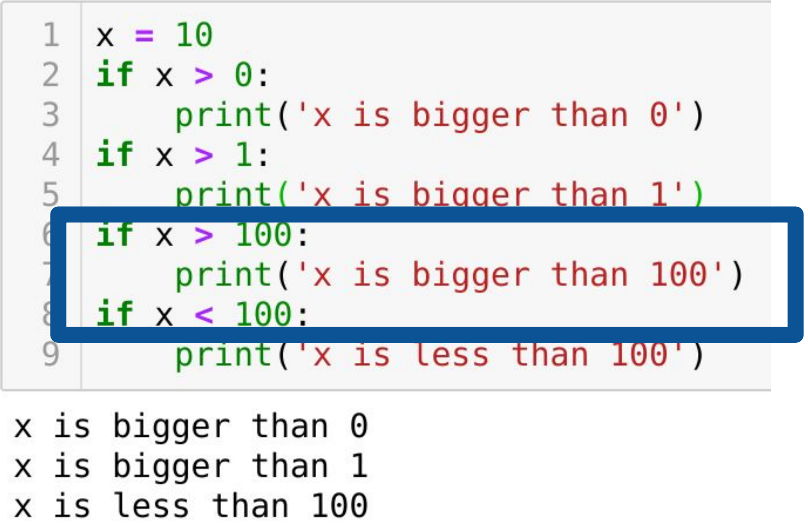
```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to `True`, then Python runs the code in the body of the if-statement.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



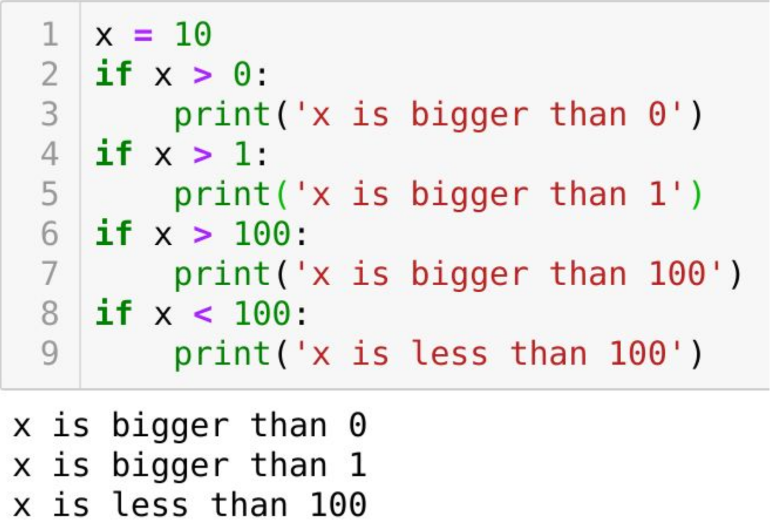
```
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to `False`, then Python skips the body and continues running code starting at the end of the if-statement.

Conditional Expressions

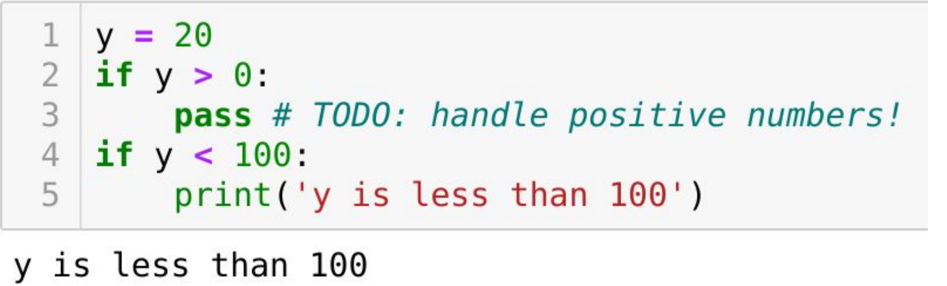
Sometimes we want to do different things depending on certain conditions



```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

Note: the body of a conditional statement can have any number of lines in it, but it must have at least one line. To do nothing, use the `pass` keyword.

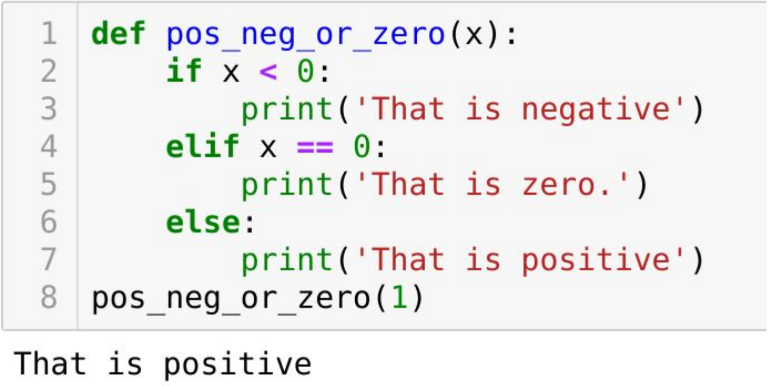


```python
y = 20
if y > 0:
    pass # TODO: handle positive numbers!
if y < 100:
    print('y is less than 100')
```

```
y is less than 100
```

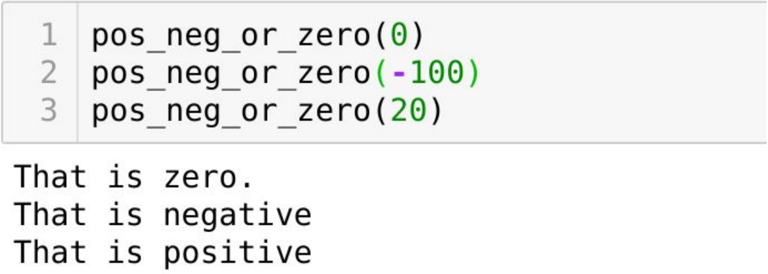
Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```python
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

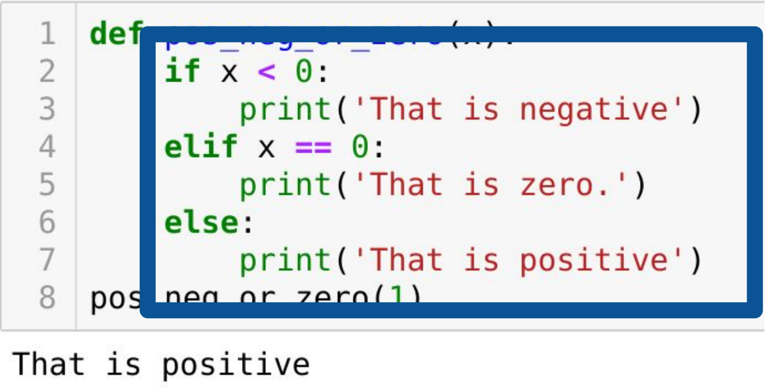


```python
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

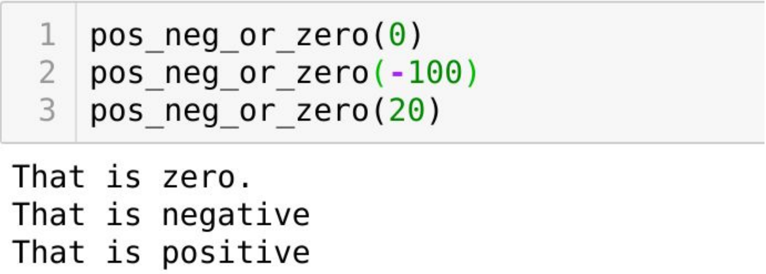
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

This is treated as a single if-statement.

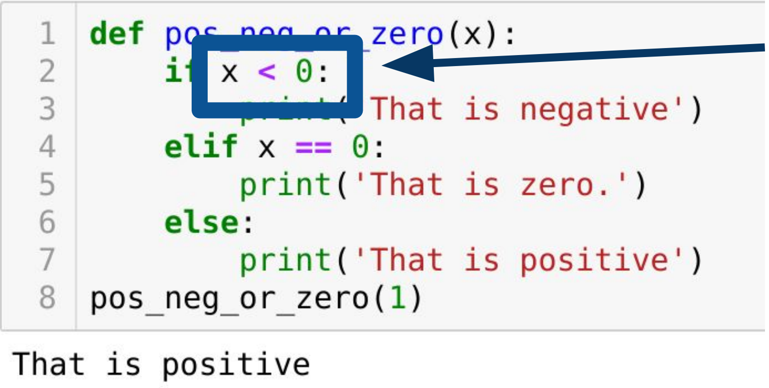


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

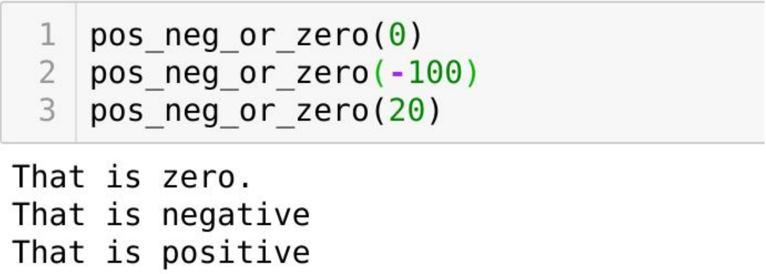
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

If this expression evaluates to `True`...

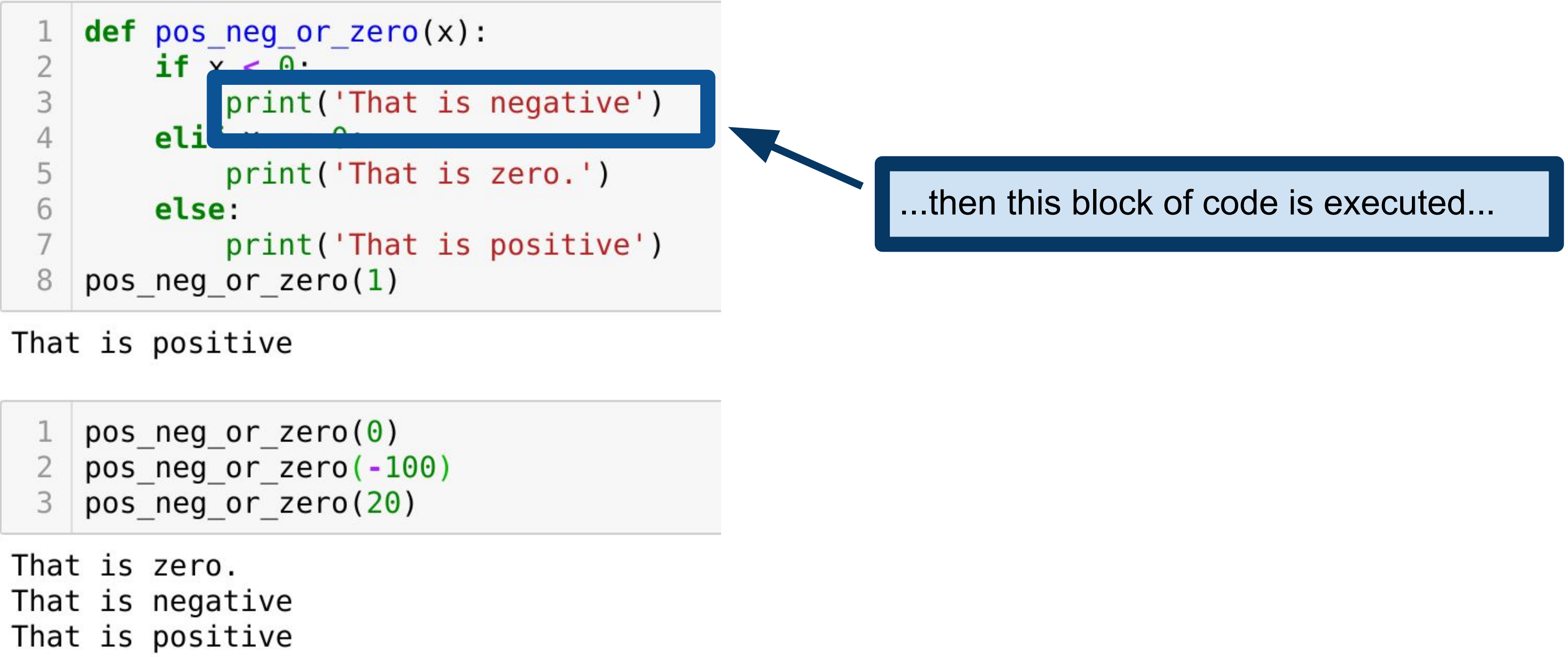


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

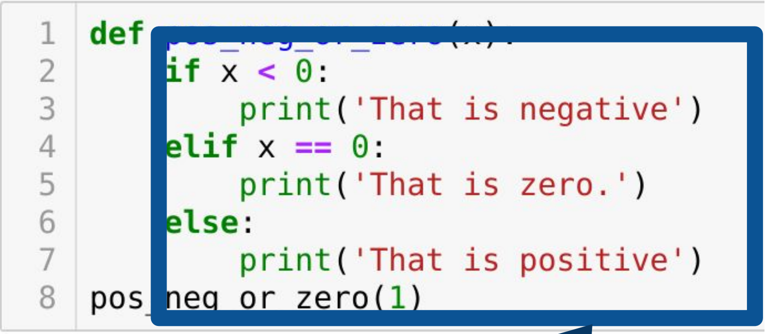
```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

...then this block of code is executed...

Conditional Expressions

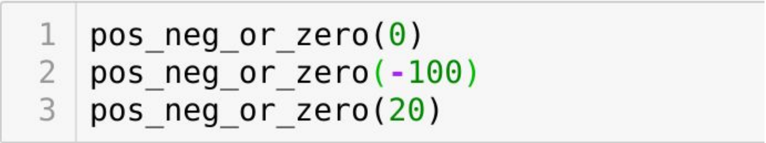
More complicated logic can be handled with **chained conditionals**



```python
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

...and then Python exits the if-statement

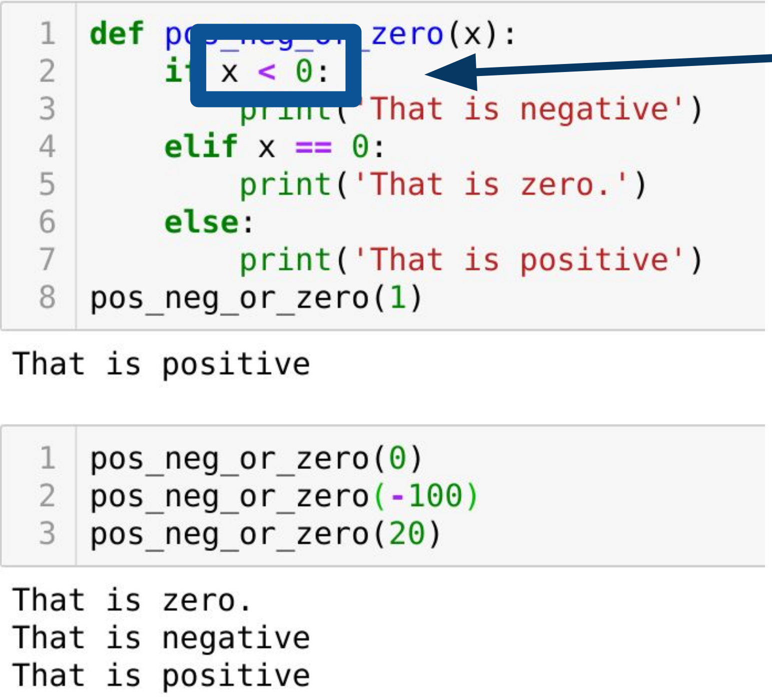


```python
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```python
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

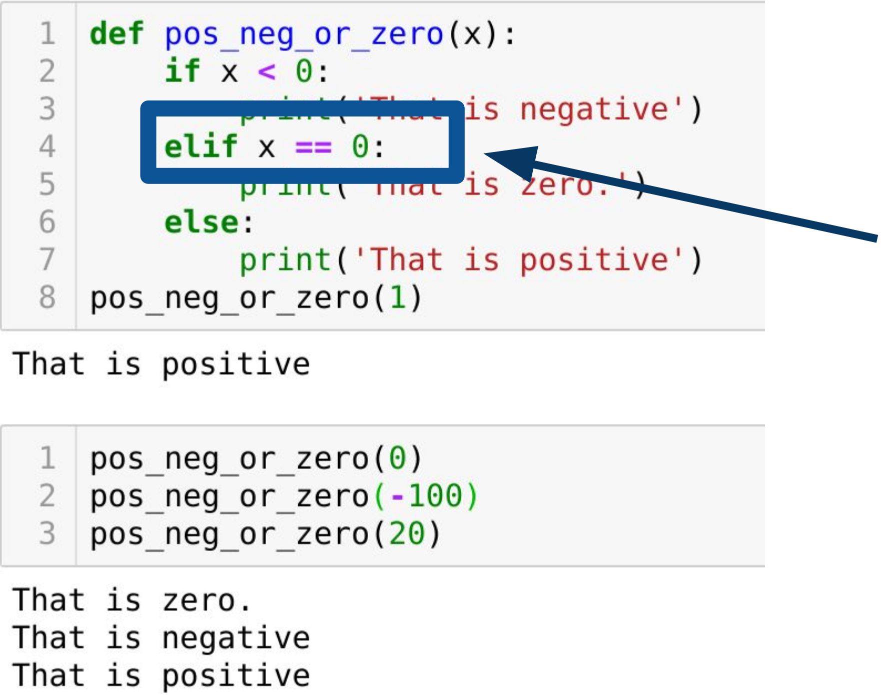
```python
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

If this expression evaluates to `False`...

Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

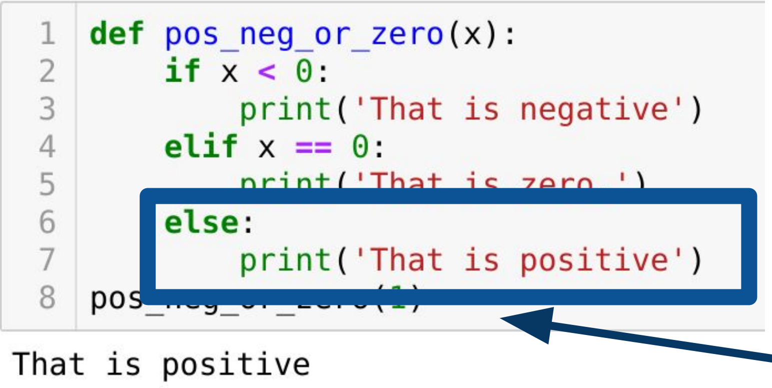
```
That is zero.
That is negative
That is positive
```

Note: `elif` is short for **else if**.

...then we go to the condition. If this condition fails, we go to the next condition, etc.

Conditional Expressions

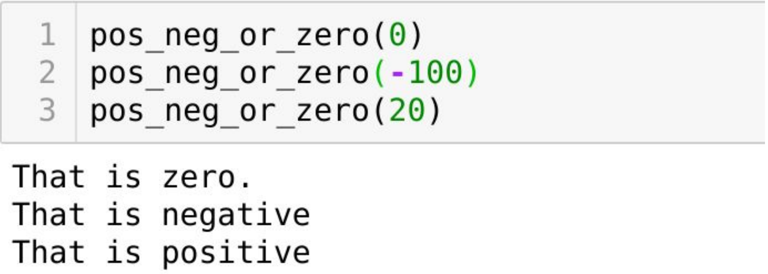
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

If all the other tests fail, we execute the block in the `else` part of the statement.

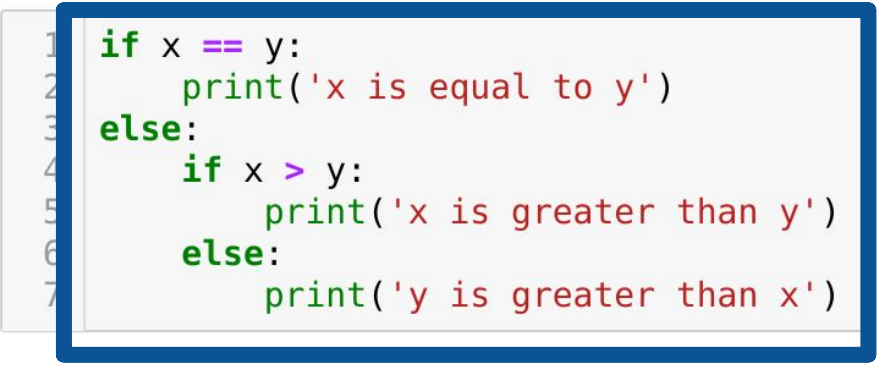


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

Conditionals can also be nested

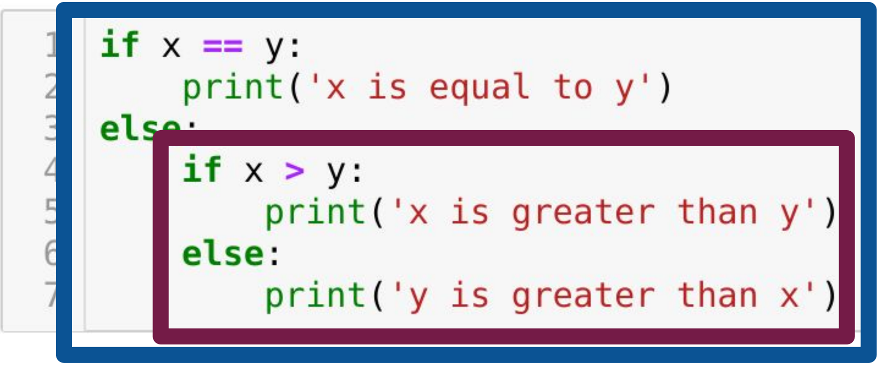


```python
if x == y:
    print('x is equal to y')
else:
    if x > y:
        print('x is greater than y')
    else:
        print('y is greater than x')
```

This if-statement...

Conditional Expressions

Conditionals can also be nested



```python
if x == y:
    print('x is equal to y')
else:
    if x > y:
        print('x is greater than y')
    else:
        print('y is greater than x')
```

This if-statement...

...contains another if-statement.

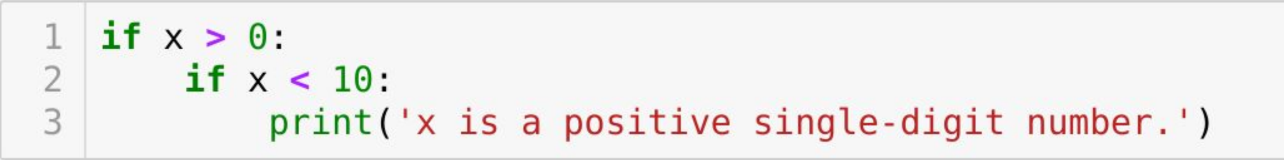
Conditional Expressions

Often, a nested conditional can be simplified

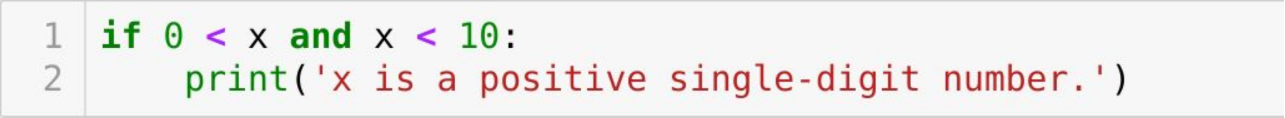
When this is possible, I recommend it for the sake of your sanity, because debugging complicated nested conditionals is tricky!

These two if-statements are equivalent, in that they do the same thing!

But the second one is (arguably) preferable, as it is simpler to read.



```python
if x > 0:
    if x < 10:
        print('x is a positive single-digit number.')
```



```python
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Week 1 practice problem

You’ll find them on Canvas in “Files/in-class practice/”

First try it on your own, then we’ll discuss it

Things to do very soon:

Install Python 3.8 and install jupyter

Familiarize yourself with jupyter:
https://jupyter.readthedocs.io/en/latest/content-quickstart.html

Read Ch 1--6 in Python 4 Everybody

Try out Google colab with your umich.edu account

Note: We will use only Python 3.8 in this course. Check that you have Python 3.8 installed on your machine and that it is running properly.

Other things

HW1 and HW2 are posted on Canvas. **Get started now!**

If you run into trouble, attend GSI office hours for help.

- Also please post to the Canvas discussion board
- If you’re having trouble, at least one of your classmates is, too
- You’ll learn more by explaining things to each other than by reading stackexchange posts!