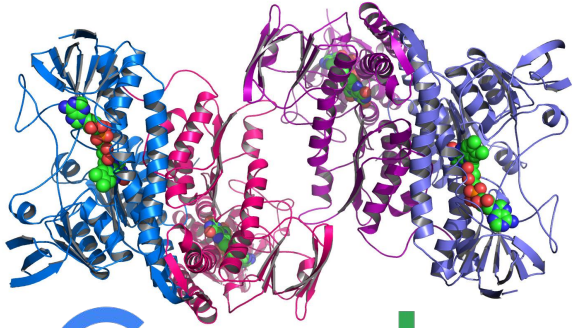
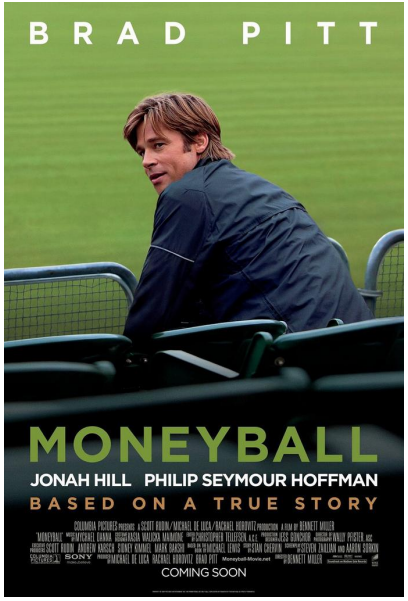
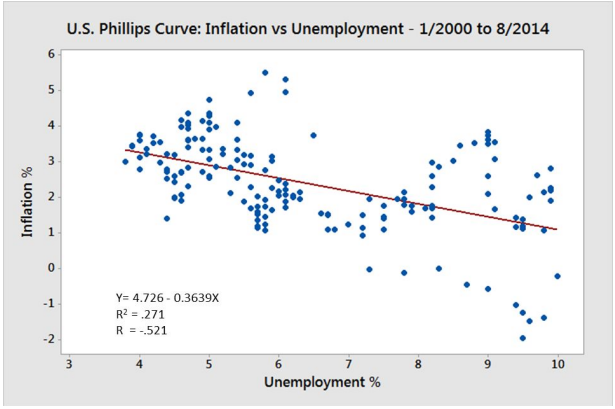
STATS 507
Data Analysis in Python

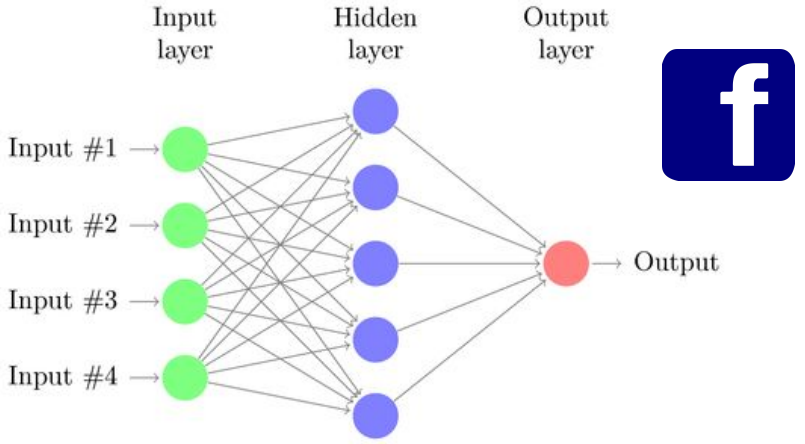
Lecture 0: Introduction and Administrivia

“Data science” has completely changed our world









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 new tools up and running
- Learn basic distributed computing frameworks

These tools will be obsolete some day...

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

Course structure

Unit 1: Introduction to Python

Data types, functions, Jupyter, classes, objects, functional programming

Unit 2: Numerical Computing and Data Visualization

numpy, scipy, matplotlib

Unit 3: Dealing with structured data

regular expressions, retrieving web data, SQL, Python pandas, APIs

Unit 4: Big data and parallel programming

Basics of the UNIX command line, ssh, Hadoop, Spark, TensorFlow

Schedule (tentative) and other information available on course webpage:

www.umich.edu/~klevin/teaching/Winter2019/STATS507

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

Course information

Instructor: Keith Levin

- Email: `klevin@umich.edu`
- Office: 256 WH
- OH: Wed 4-5:30
  or by appointment

GSI: Roger Fan

- Email: `rogerfan@umich.edu`
- OH: TBA

Textbook: None

- Readings posted to the website

Grading: 8-10 HWs,

- Weighted approximately equally
- No midterm, no final
- No class project
- Late days (see syllabus)

See syllabus on Canvas or at
umich.edu/~klevin/teaching/Fall2019/STATS507/syllabus.pdf

A Note on Enrollment and the Waitlist

This is an immensely popular course...

… which is excellent, but it means that there are… a lot of you.

Waitlist:

The waitlist is handled by the statistics office. I have no control of it!

Please do not email me asking for overrides. I cannot grant them.

Please direct all enrollment questions to the statistics office: **stat-um@umich.edu**

Before we continue...

Readings:

For the first half of the course, readings will be given in both

Allen B. Downey's *Think Python* and

Charles Severance's *Python for Everybody*

You can do the readings out of either one, whichever you prefer!

Later, we'll make exclusive use of Severance

A Note on Readings

I will post weekly readings throughout the course

I would prefer if you do the readings before lecture...

...but I recognize this is not always possible...

...and if you find that you learn better seeing lecture first, then that's fine.

Some of the readings consist of technical documentation

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…

...and it's okay to set things aside to come back to later!

Policies

Don’t plagiarize!

- You may discuss homeworks with your fellow students...
- ...but you must submit your own work
- Disclose in your homework whom (if anyone) you worked with

Late homeworks are not allowed!

- Instead, we have “late days”, of which you get 7
- One late day extends HW deadline by 24 hours
- **Note:** homework deadlines may not be extended beyond 11:59pm on the scheduled day of the final (Friday, December 20th).

Refer to the syllabus for details.

Survey time!

1. Raise your hand if you have used Python before.
2. Raise your hand if you have used jupyter/iPython in the past.
3. Raise your hand if you have used the UNIX/Linux command line.
4. Raise your hand if you have used the Python `matplotlib` package.
5. Raise your hand if you prefer Canvas over a course webpage

Things to do very soon:

Pick an editor/IDE for python

or just use a text editor, or just write directly in jupyter

Familiarize yourself with jupyter:

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

Get a Cavium username

You should have gotten an email that a username has been created for you

If you are enrolled in the course and do not have a username:

Fill out a form here: http://myumi.ch/6pn5d, list me as your “advisor”

Note: we will use only Python 3 in this course. Check that you have Python 3 installed on your machine and that it is running properly.

Other things

HW1 is posted to canvas and the website. **Get started now!**

If you run into trouble, come to office hours for help

- But also please post to the discussion board on Canvas
- 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!

Email policy:

I will **not** provide tech support over email!

If you are having trouble, post to the discussion board and/or come to OHs!

STATS 507
Data Analysis in Python

Lecture 1: Introduction to Python

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 on your computer

- Python interpreter
- Jupyter: https://jupyter.org/
- PythonAnywhere: https://www.pythonanywhere.com/
- Suggestions from Allen Downey:
  - http://www.allendowney.com/wp/books/think-python-2e/

Your homeworks must be handed in as Jupyter notebooks

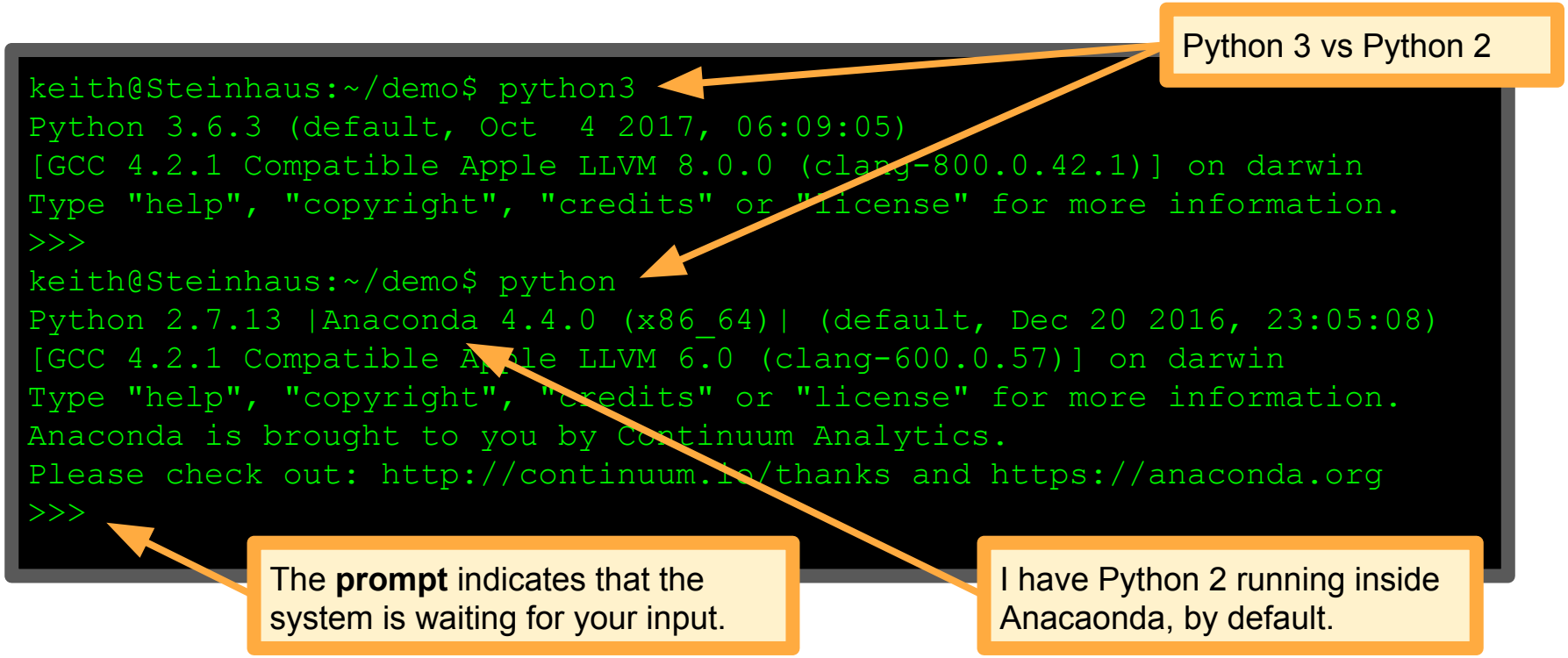
- But you should also be comfortable with the interpreter and running Python on the command line
- Installing Jupyter: https://jupyter.readthedocs.io/en/latest/install.html
  - **Note:** Jupyter recommends Anaconda: https://www.anaconda.com/
    - I mildly recommend against Anaconda, but it's your choice

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

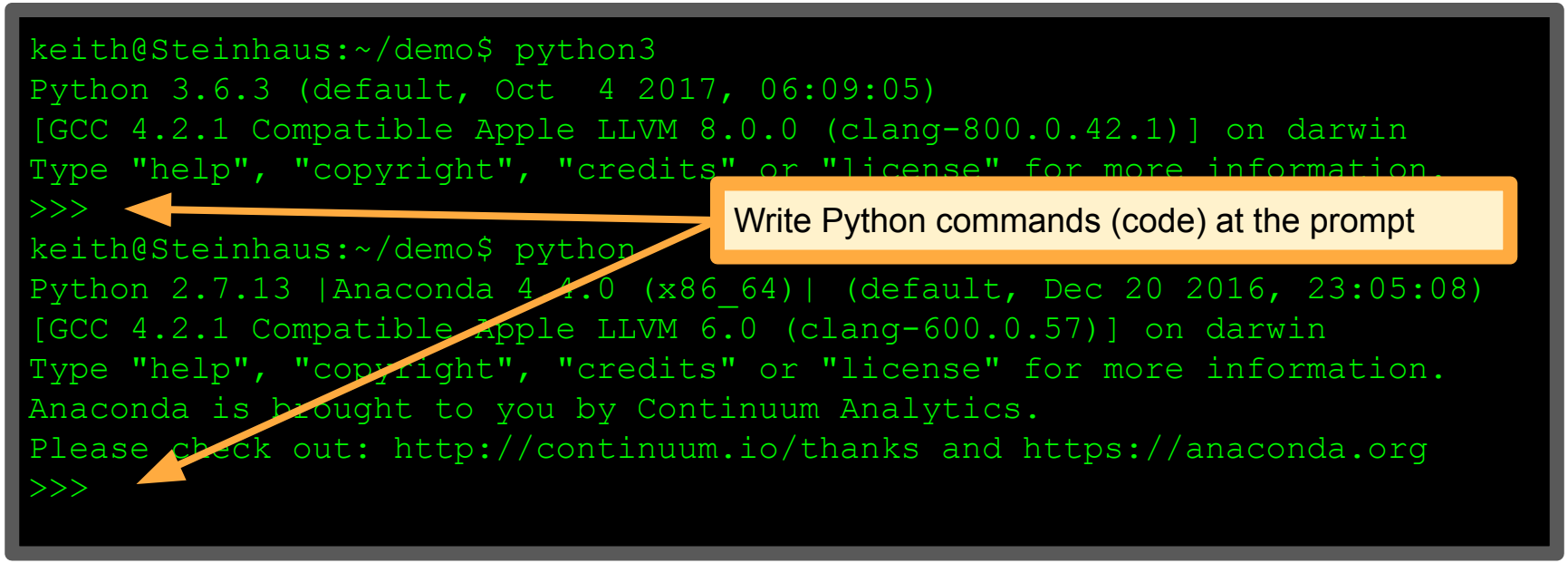
Python Interpreter on the Command Line

```
keith@Steinhaus:~/demo$ python3
Python 3.6.3 (default, Oct  4 2017, 06:09:05)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
keith@Steinhaus:~/demo$ python
Python 2.7.13 |Anaconda 4.4.0 (x86_64)| (default, Dec 20 2016, 23:05:08)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

Python Interpreter on the Command Line



Python Interpreter on the Command Line



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

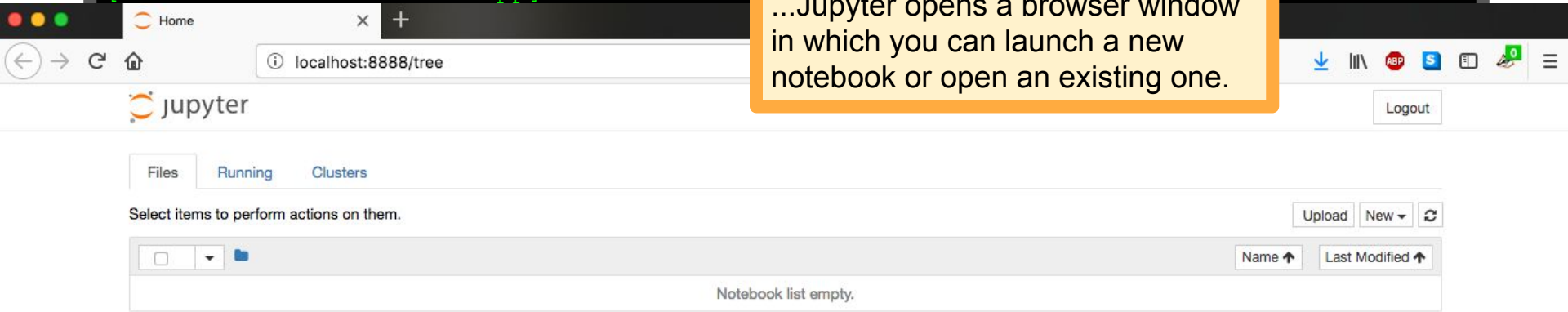
```
keith@Steinhaus:~/demo$ jupyter notebook
[I 17:11:41.129 NotebookApp] Serving notebooks from local directory:
/Users/keith/Dropbox/Academe/Teaching/STATS507/Lecs/L1_AdminIntro
[I 17:11:41.129 NotebookApp] 0 active kernels
[I 17:11:41.129 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=452d6d4b227f306f5bb57e72f5d4722fcbadf47d1d794441
[I 17:11:41.129 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
[C 17:11:41.132 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
    http://localhost:8888/?token=452d6d4b227f306f5bb57e72f5d4722fcbadf47d1d794441
[I 17:11:41.635 NotebookApp] Accepting one-time-token-authenticated connection from
::1
```

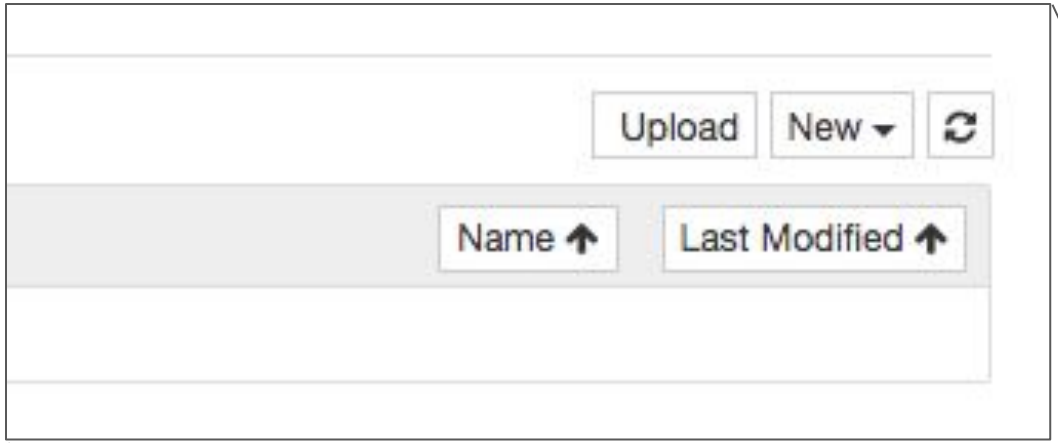
Jupyter provides some information about its startup process, and then...

Running Jupyter

```
keith@Steinhaus:~/demo$ jupyter notebook
[I 17:11:41.129 NotebookApp] Serving notebooks from local directory:
/Users/keith/Dropbox/Academe/Teaching/STATS507/Lecs/L1_AdminIntro
[I 17:11:41.129 NotebookApp] 0 active kernels
[I 17:11:41.129 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=452d6d4b227f306f5bb57e72f5d4722fcbadf47d1d794441
[I 17:11:41.129 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
[C 17:11:41.132 NotebookApp]
```

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





jupyter

Home

localhost:8888/tree

Search

jupyter

Logout

Files Running Clusters

Select items to perform actions on them.

Upload New

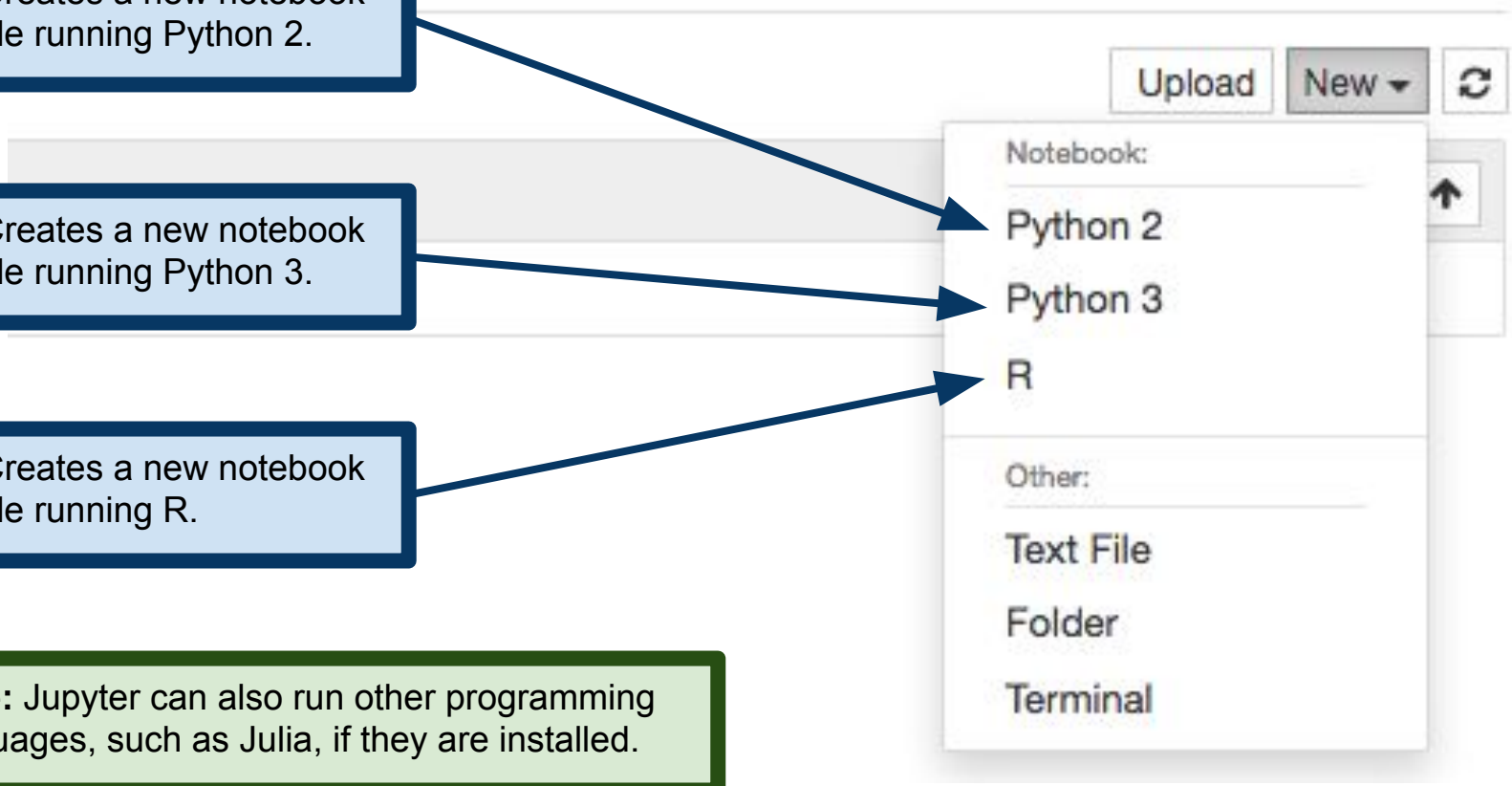
Name Last Modified

Notebook list empty.

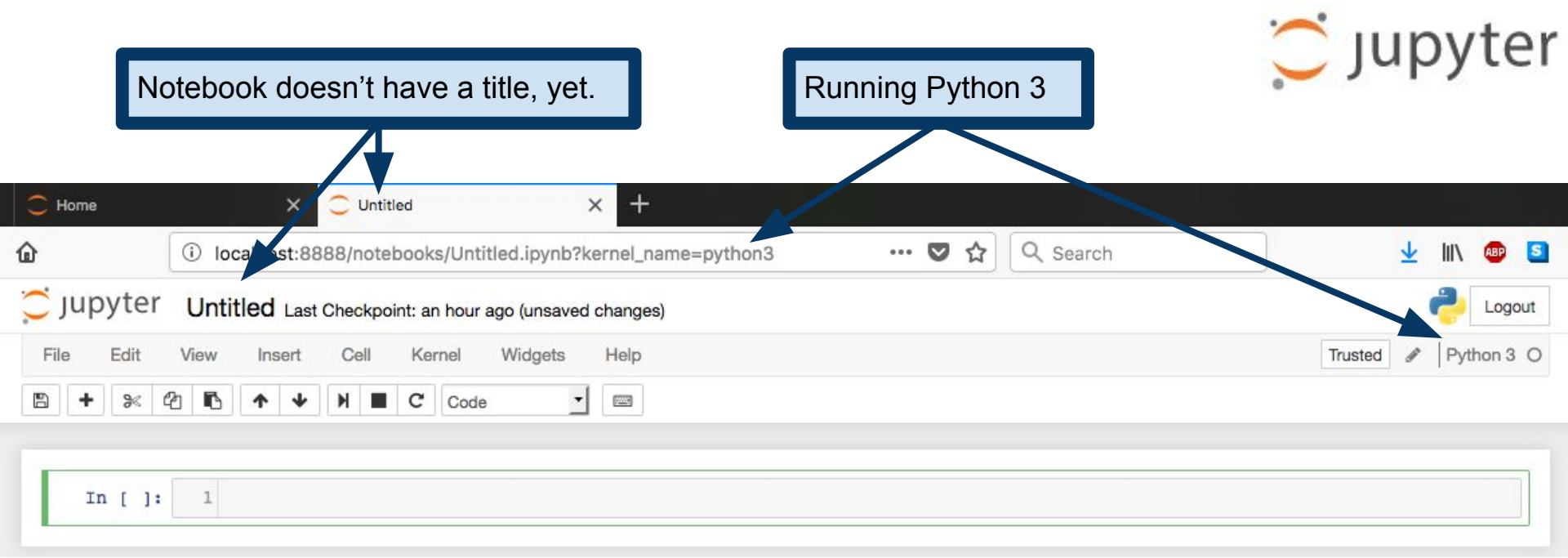
Creates a new notebook file running Python 2.

Creates a new notebook file running Python 3.

Creates a new notebook file running R.

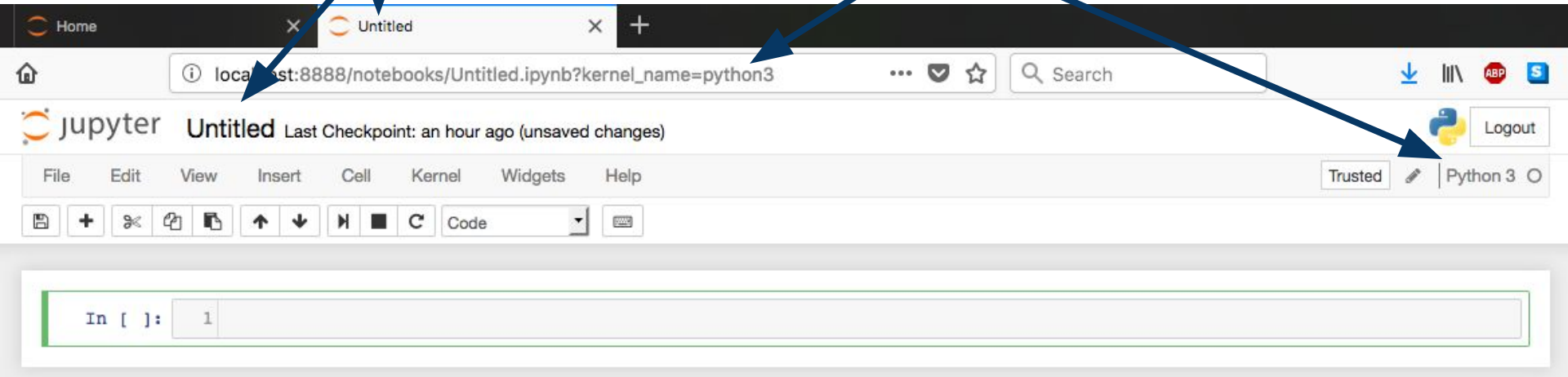


Note: Jupyter can also run other programming languages, such as Julia, if they are installed.


Notebook doesn't have a title, yet.
Running Python 3
jupyter
Home
Untitled
Search
jupyter
Untitled
Last Checkpoint: an hour ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Code
In [ ]:
1

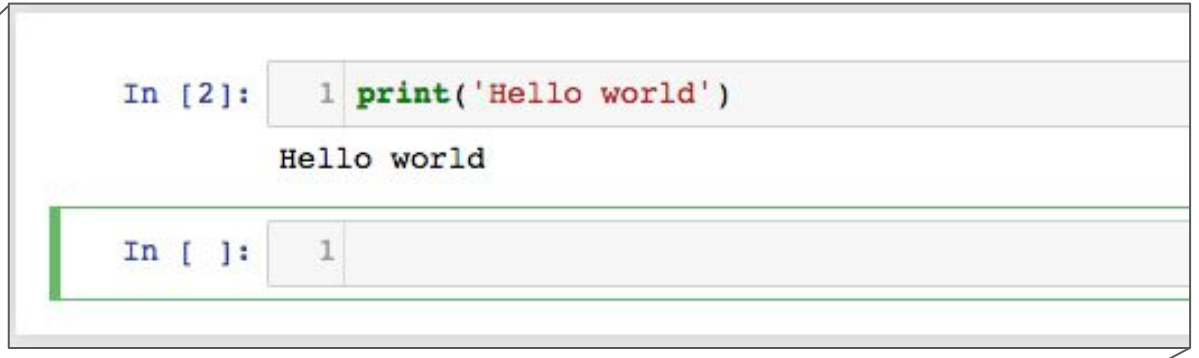
Notebook doesn't have a title, yet.

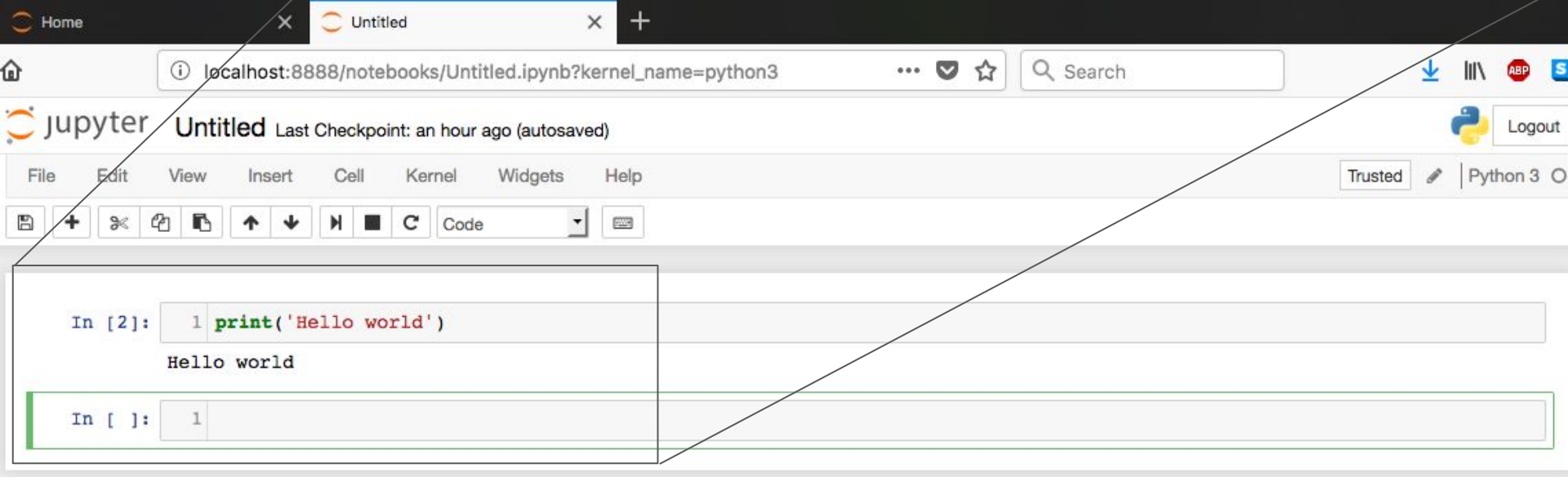
Running Python 3



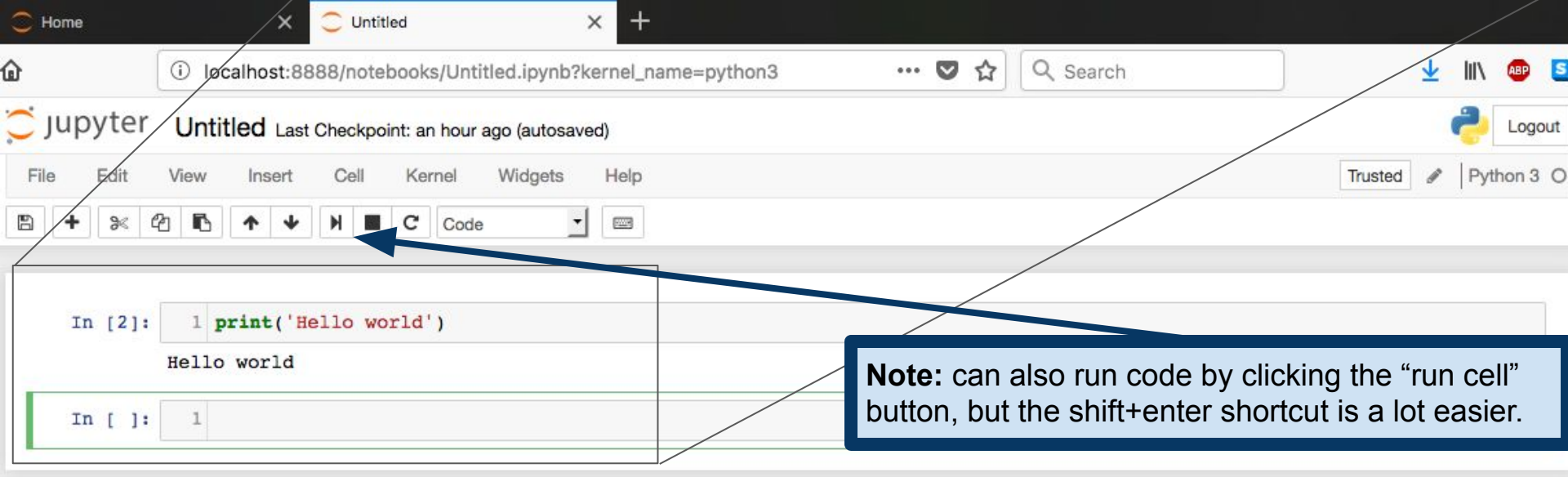
I'll leave it to you to learn about the other features by reading the documentation. For now, the green-highlighted box is most important. That's where we write Python code.

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





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



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 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

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

```
Hello world
```

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

```
Hello world!
```

Arithmetic in Python

```
1 1+2
```

```
3
```

Use + to add numbers.

```
1 2*3
```

```
6
```

Use * to multiply.

```
1 2*3 - 1
```

```
5
```

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

```
1 2**7
```

```
128
```

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

/ for division.

```
1 6/3
```

```
2.0
```

```
1 8/3
```

```
2.6666666666666665
```

// performs division but rounds down.

```
1 8//3
```

```
2
```

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

```
1 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**

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

Assign values to three variables

```
1 mystring
```

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

```
1 number_of_planets
```

```
9
```

```
1 number_of_planets = 8
2 number_of_planets
```

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

```
8
```

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**

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

Assign values to three variables

```
1 mystring
```

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

```
1 number_of_planets
```

```
9
```

```
1 number_of_planets = 8
2 number_of_planets
```

```
8
```

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

Variables in Python

Variable is a name that refers to a value

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

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

```
1 mystring
```

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

```
1 number_of_planets
```

```
9
```

```
1 number_of_planets = 8
2 number_of_planets
```

```
8
```

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.

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:

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

Variables in Python

Sometimes we do need to know the type of a variable

Python `type()` function does this for us

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

```
str
```

```
1 type(approx_pi)
```

```
float
```

```
1 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`:

```
1 approx_pi = 3.141592
2 type(approx_pi)
```

```
float
```

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

```
int
```

```
1 pi_int
```

```
3
```

Convert a `string` to an `int`:

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

```
int
```

```
1 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`:

```
1 approx_pi = 3.141592
2 type(approx_pi)
```

```
float
```

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

```
int
```

```
1 pi_int
```

```
3
```

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

Convert a `string` to an `int`:

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

```
int
```

```
1 int_from_str
```

```
8675309
```

```
1 float_from_int = float(42)
2 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`:

```
1 approx_pi = 3.141592
2 type(approx_pi)
```

```
float
```

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

```
int
```

```
1 pi_int
```

```
3
```

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

Convert a `string` to an `int`:

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

```
int
```

```
1 int_from_str
```

```
8675309
```

```
1 float_from_int = float(42)
2 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...

```
1 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

```
  1 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

```
1 '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

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

```
'catdog'
```

```
1 '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!

```
1 # This is a comment.
2 # Python doesn't try to run code that is
3 # "commented out".
4 euler = 2.71828 # Euler's number
5 '''Triple quotes let you write a multi-line comment
6     like this one. Everything between the first
7     triple-quote and the second one will be ignored
8     by Python when you run your program'''
9 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

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

```
1.41421356237
```

```
1 a=2
2 b=3
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 "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

```
1 import math
2 rt2 = math.sqrt(2)
3 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.

```
1 a=2
2 b=3
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 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

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

```
1.41421356237
```

```
1 a=2
2 b=3
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

```
1  a = 60
2  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.

```
1  x = 1.71828
2  y = math.exp( -math.log(x+1))
3  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

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

```
1  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

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

```
1 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

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

```
1  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

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

```
1  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

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

```
1  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

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

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

```
1  print_wittgenstein()
```

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

Defining Functions

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

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

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

```
1  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

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

```
1  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

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

```
1  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

```
1 def wittgenstein_sandwich(bread):
2     print(bread)
3     print_wittgenstein()
4     print(bread)
5 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

```
1 def wittgenstein_sandwich(bread):
2     print(bread)
3     print_wittgenstein()
4     print(bread)
5 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

```
1 def wittgenstein_sandwich(bread):
2     print(bread)
3     print_wittgenstein()
4     print(bread)
5 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

```
1 def wittgenstein_sandwich(bread):
2     print(bread)
3     print_wittgenstein()
4     print(bread)
5 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

```
1 def wittgenstein_sandwich(bread):
2     print(bread)
3     print_wittgenstein()
4     print(bread)
5 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

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

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

```
'birdbird'
```

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

```
1 print(twogoats)
```

```
goatgoat
```

Defining Functions

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

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

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

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

```
'birdbird'
```

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

```
1 print(twogoats)
```

```
goatgoat
```

Defining Functions

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

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

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

```
'birdbird'
```

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

```
1 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

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

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

```
'birdbird'
```

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

```
1 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

```
1 def wittgenstein_sandwich(bread):
2     local_var = 1 # define a useless variable, just as example.
3     print(bread)
4     print_wittgenstein()
5     print(bread)
6 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.

```
1 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

```
1 type(print_wittgenstein)
```

```
function
```

```
1 print_wittgenstein
```

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

```
1 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.

A parting note for the day...

Homework:

Start your homework early!

If you run into technical issues, you'll want to have time to come get help!

A note on pace and difficulty

I aim to teach Python from scratch in this course, but…

...time spent on Python is time not spent on the stuff you're really here for

So, I expect that you are willing to work hard to keep up

If I am moving too fast, or you don't understand something, come speak to me promptly!