

DATA SOCIETY®

Introduction to Python - Day 1

*"One should look for what is and not what he thinks should be."
-Albert Einstein.*

Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	
Use Anaconda and Jupyter Notebook to access and write code	
Install all packages that will be needed for the day	
Work with variables	
Summarize best practices for writing code	
Define various data types and perform basic operations	
Identify and create basic data structures	

DEA pain pill database

- We're going to look at a dataset maintained by the Drug Enforcement Administration that tracks the path of every single pain pill sold in the United States, by manufacturers and distributors to pharmacies in every town and city has been made public.
- Create a free account with Washington Post to view the dataset

https://www.washingtonpost.com/graphics/2019/investigations/dea-pain-pill-database/?utm_term=.3140d17eb10c#download-resources

What is Python?



- **Python** is a powerful programming language that data scientists love because of its:
 - inherent readability and simplicity compared to lower level languages
 - number of dedicated analytical libraries to work with numerical, text, and image data
 - Over 72,000 libraries and growing constantly
 - [Click here](#) for more about Python

What can you do with Python?


Python use cases relating to data and analytics

- Sentiment analysis using natural language processing
- Twitter analysis using live Twitter feeds
- Object recognition using deep learning
- Facial recognition using deep learning
- Web scraping to automate the collection of data from websites
- Interactive visualization deployable to websites
- 3D visualizations
- Machine learning on structured data
- Data ingestion from various sources
- Data wrangling with fast and efficient functions
- Big Data modeling through integration with Apache Spark

Python vs. R

	Python	R
Learning curve	Steeper learning curve for people without a programming background	Can be easy to learn for people without a programming background
Automation	Once you write commands you won't have to re-do the work, just upload a new data set	Once you write commands you won't have to re-do the work, just upload a new data set
Analyzing data	Lots of libraries contributed by a broad user community	Over 6,500 packages contributed by the community including top academics
Speed	In-memory, only limited by your computer's RAM	In-memory, only limited by your computer's RAM
Type of data	Reads data of almost any type	Reads data of almost any type
Compatibility	Compatible with almost any data output, storage or processing platform	Not as compatible as Python with some data architecture systems, may need custom-built interfaces

Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	
Use Anaconda and Jupyter Notebook to access and write code	
Install all packages that will be needed for the course	
Work with variables	
Summarize best practices for writing code	
Define various data types and perform basic operations	
Identify and create basic data structures	

What is Anaconda?

- Anaconda is an **open-source distribution of Python** that is used to simplify package management and deployment
- You can install, update and remove R and other programming language packages or even packages by canonical conda



Install Anaconda

For Windows

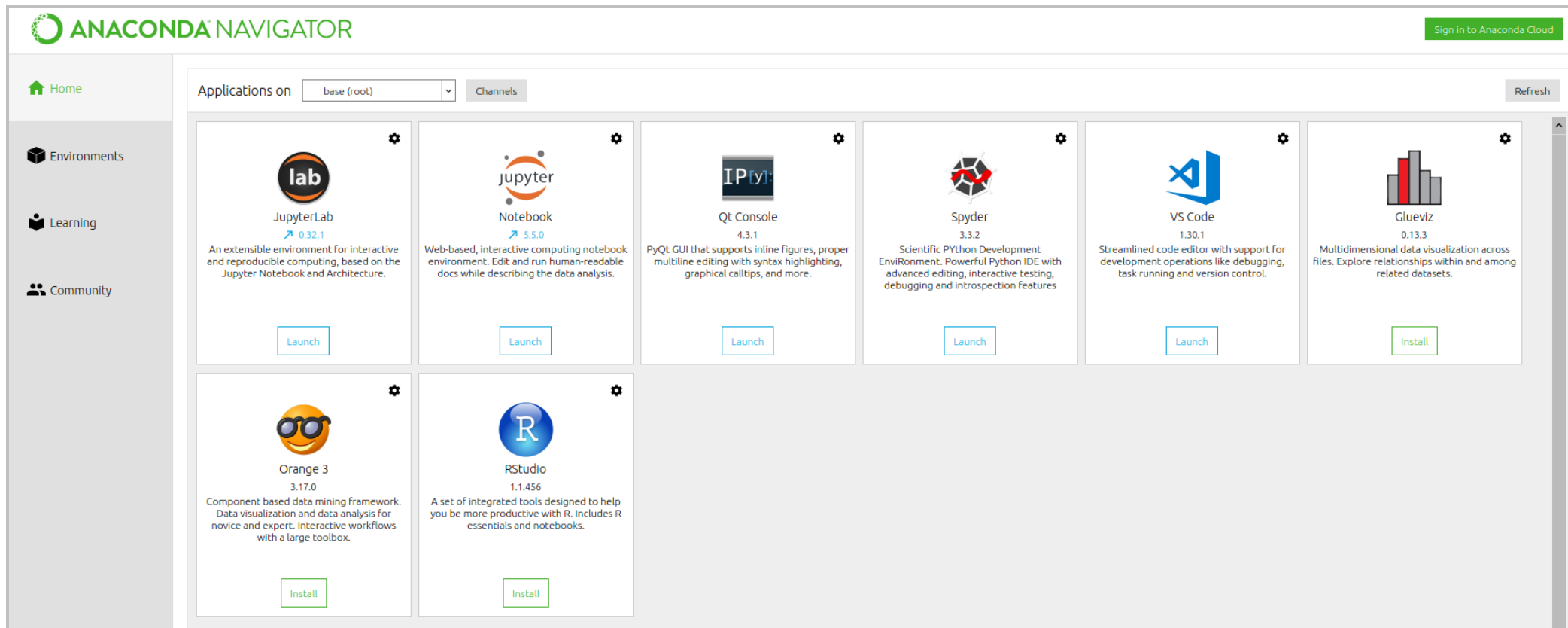
- Anaconda with Python 3.6 or greater
- Download link: https://repo.continuum.io/archive/Anaconda3-2018.12-Windows-x86_64.exe

For Mac

- Anaconda with Python 3.6 or greater
- Download link: https://repo.continuum.io/archive/Anaconda3-2018.12-MacOSX-x86_64.pkg

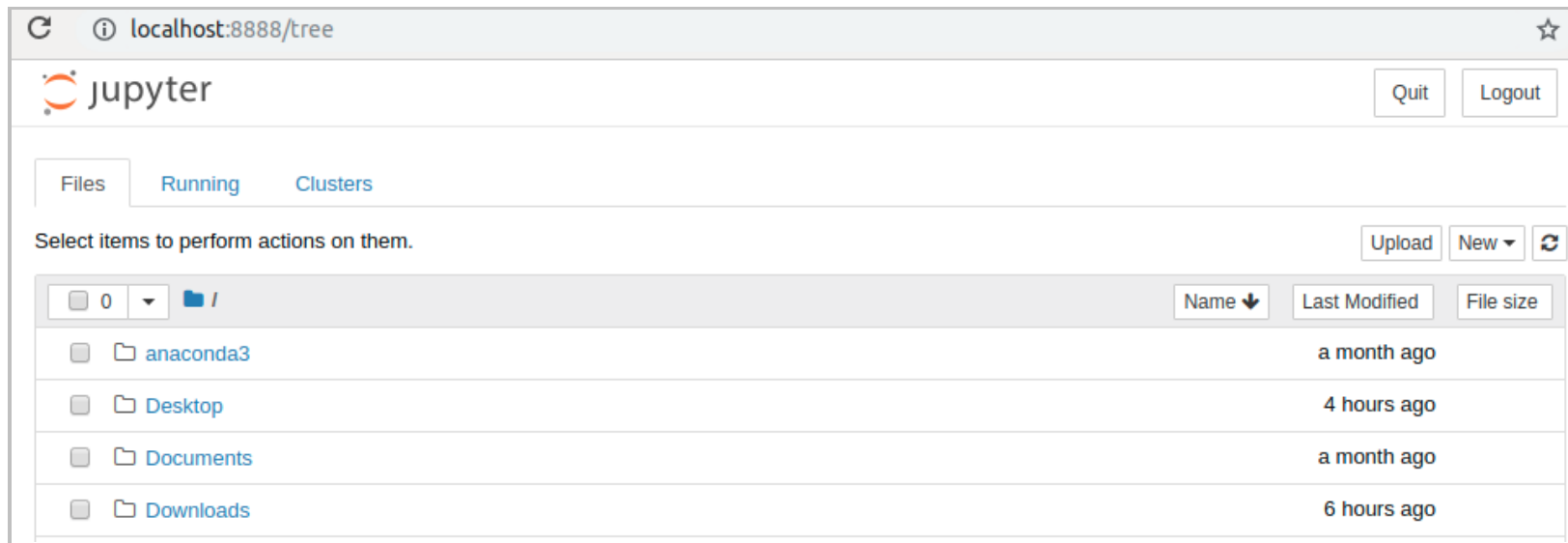
Check installed IDE for Python

- Once installed, open **Anaconda Navigator** through your application explorer and click on Home tab



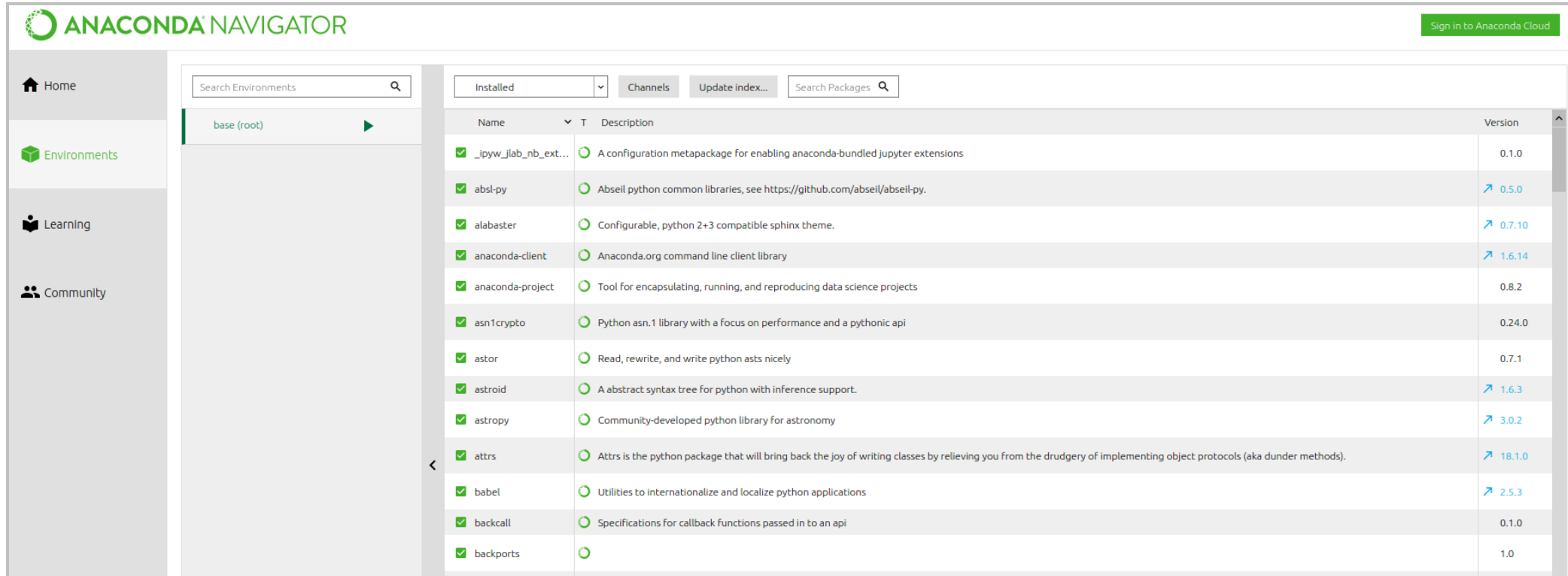
Check installed IDE for Python

- If Jupyter Notebook is not installed, click on the Install button to install it
- Then, click on Launch button to test it
- If it is working, you should see a browser window open



Check installed packages

- Now click on Environment tab

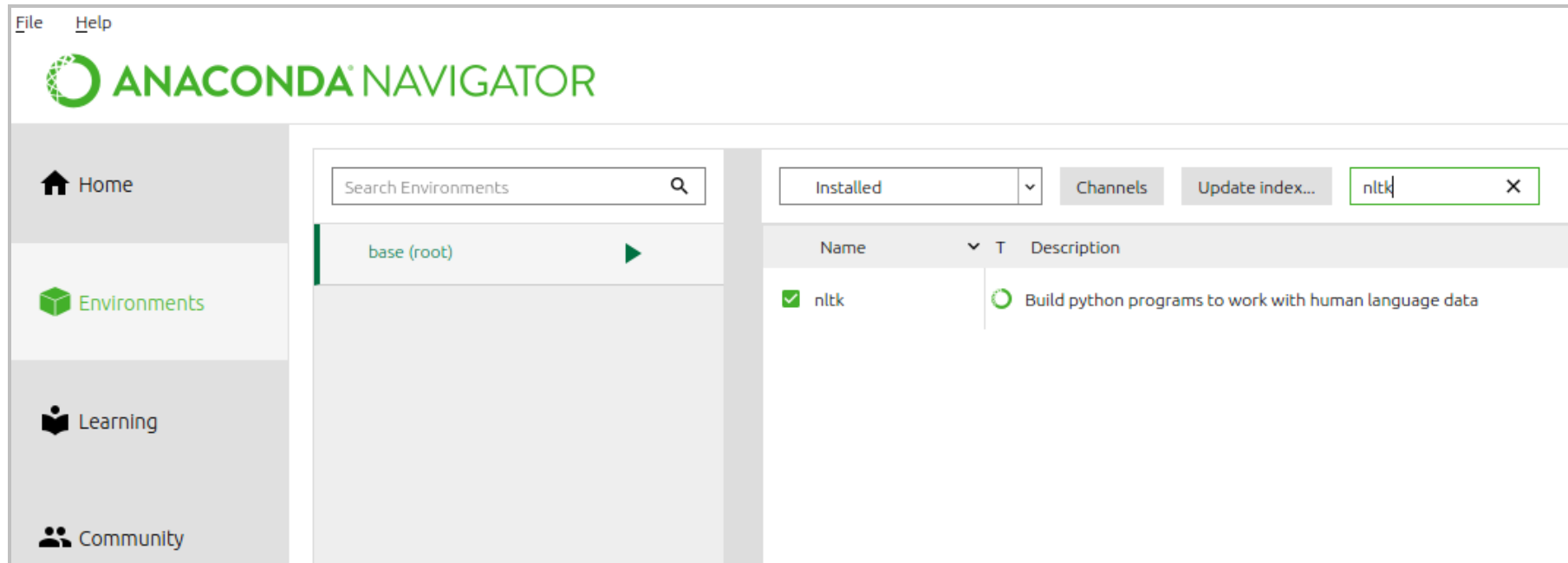


The screenshot shows the Anaconda Navigator application. The left sidebar contains navigation links: Home, Environments, Learning, and Community. The 'Environments' tab is active, showing a list of environments with 'base (root)' selected. The main panel displays the 'base (root)' environment with a table of installed packages. The table has columns for Name, Description, and Version. The 'Name' column includes a checkmark icon for installed packages. The 'Description' column provides a brief description of each package. The 'Version' column shows the installed version and a link to the latest version.

Name	Description	Version
✓ _ipyw_jlab_nb_ext...	A configuration metapackage for enabling anaconda-bundled jupyter extensions	0.1.0
✓ absl-py	Abseil python common libraries, see https://github.com/abseil/abseil-py .	0.5.0
✓ alabaster	Configurable, python 2+3 compatible sphinx theme.	0.7.10
✓ anaconda-client	Anaconda.org command line client library	1.6.14
✓ anaconda-project	Tool for encapsulating, running, and reproducing data science projects	0.8.2
✓ asn1crypto	Python asn.1 library with a focus on performance and a pythonic api	0.24.0
✓ astor	Read, rewrite, and write python asts nicely	0.7.1
✓ astroid	A abstract syntax tree for python with inference support.	1.6.3
✓ astropy	Community-developed python library for astronomy	3.0.2
✓ attrs	Attrs is the python package that will bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols (aka dunder methods).	18.1.0
✓ babel	Utilities to internationalize and localize python applications	2.5.3
✓ backcall	Specifications for callback functions passed in to an api	0.1.0
✓ backports		1.0

Check installed packages

- Select the base (root) environment (or whichever one you'll use for the class)
- Check the dropdown menu showing Installed packages to see packages already installed with your Anaconda distribution
- To check if a package is installed, type in the name of the package in the Search bar

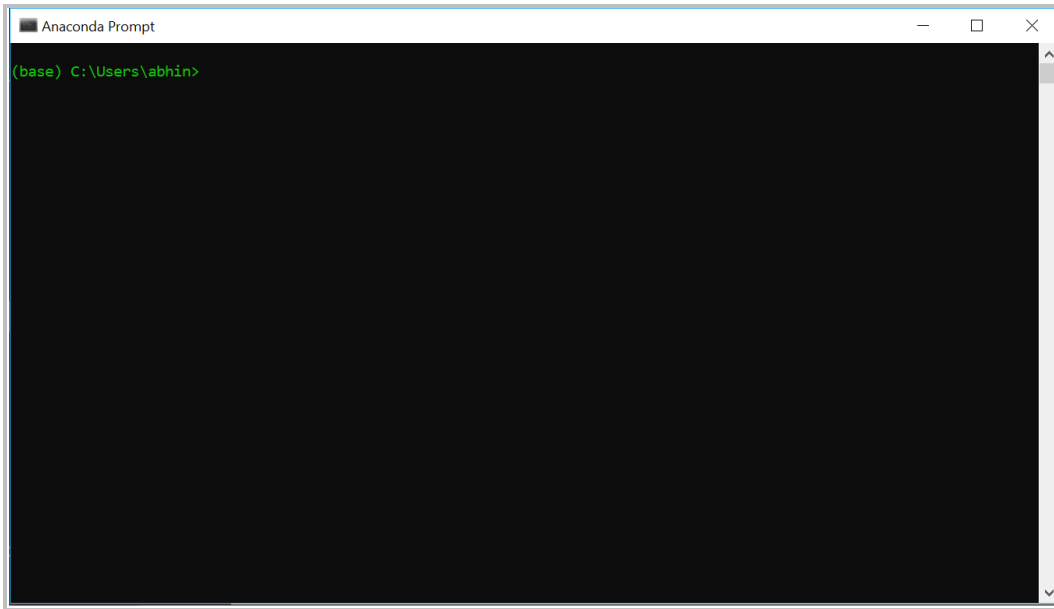


- **NOTE:** package os is already installed with your base Python, but you won't be able to look it up through Anaconda Navigator, simply skip it!

Install packages

For Windows

- Open **Anaconda Prompt** from your application explorer



For Mac

- Open your **Terminal**



Install packages

- To install a package, type a command following this template

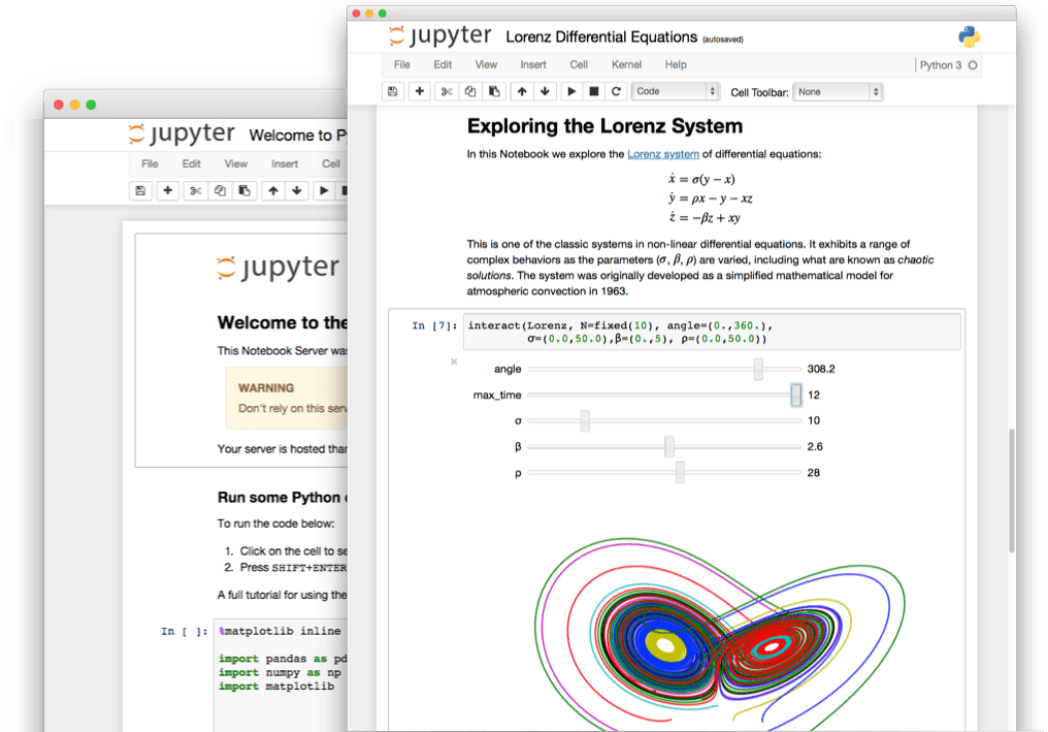
```
conda install [package_name]
```

- For example, if you need to install `nltk`, your command would look like this:

```
conda install nltk
```

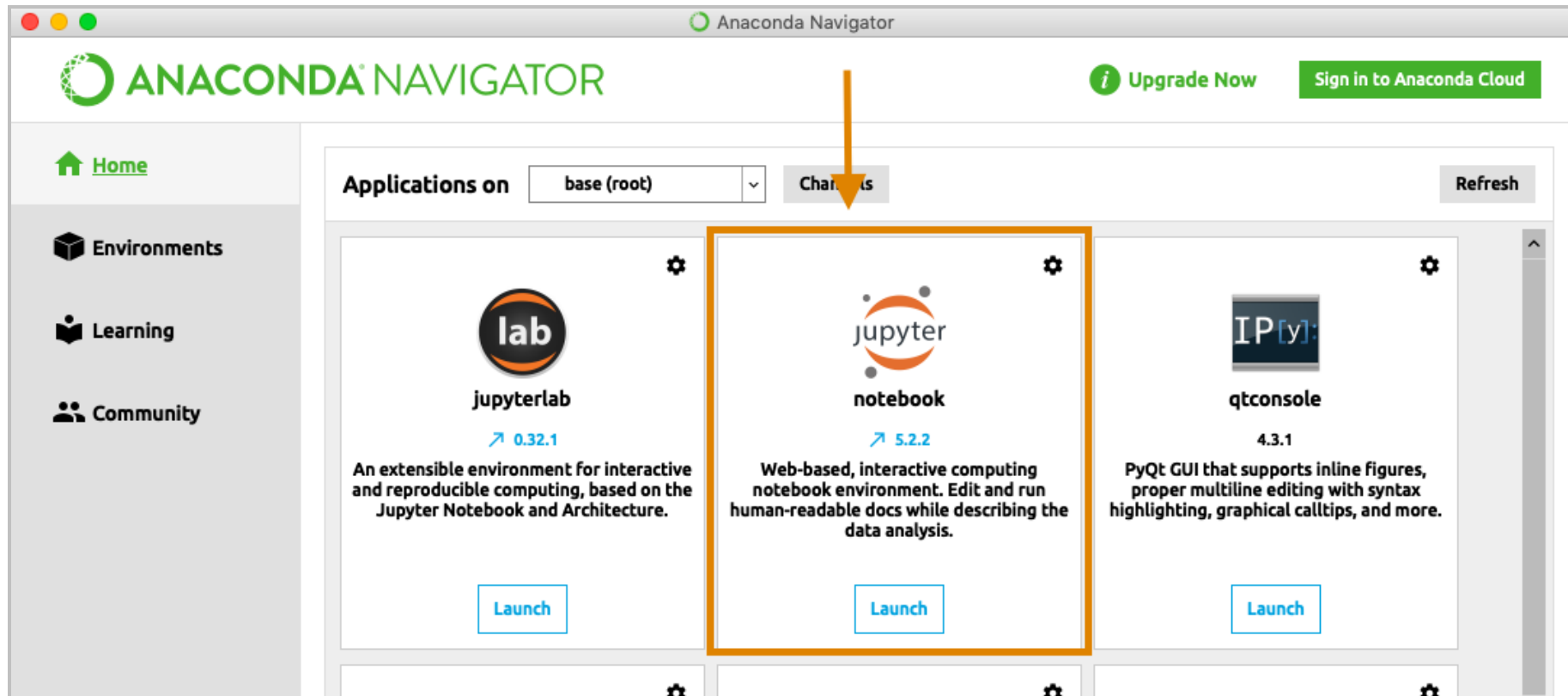
What is a Jupyter Notebook?

- Jupyter Notebook is an **Interactive Development Environment (IDE)** that comes with Anaconda
- It is an **open-source, web-based IDE** with deep cross-language integration which allows you to share documents containing live code, equations, visualizations and narrative text
- Data scientists love using Jupyter for data cleaning and transformation, statistical modeling, visualization, machine learning, deep learning, and more
- The format Jupyter Notebook uses is `ipynb` which has become an **industry standard**



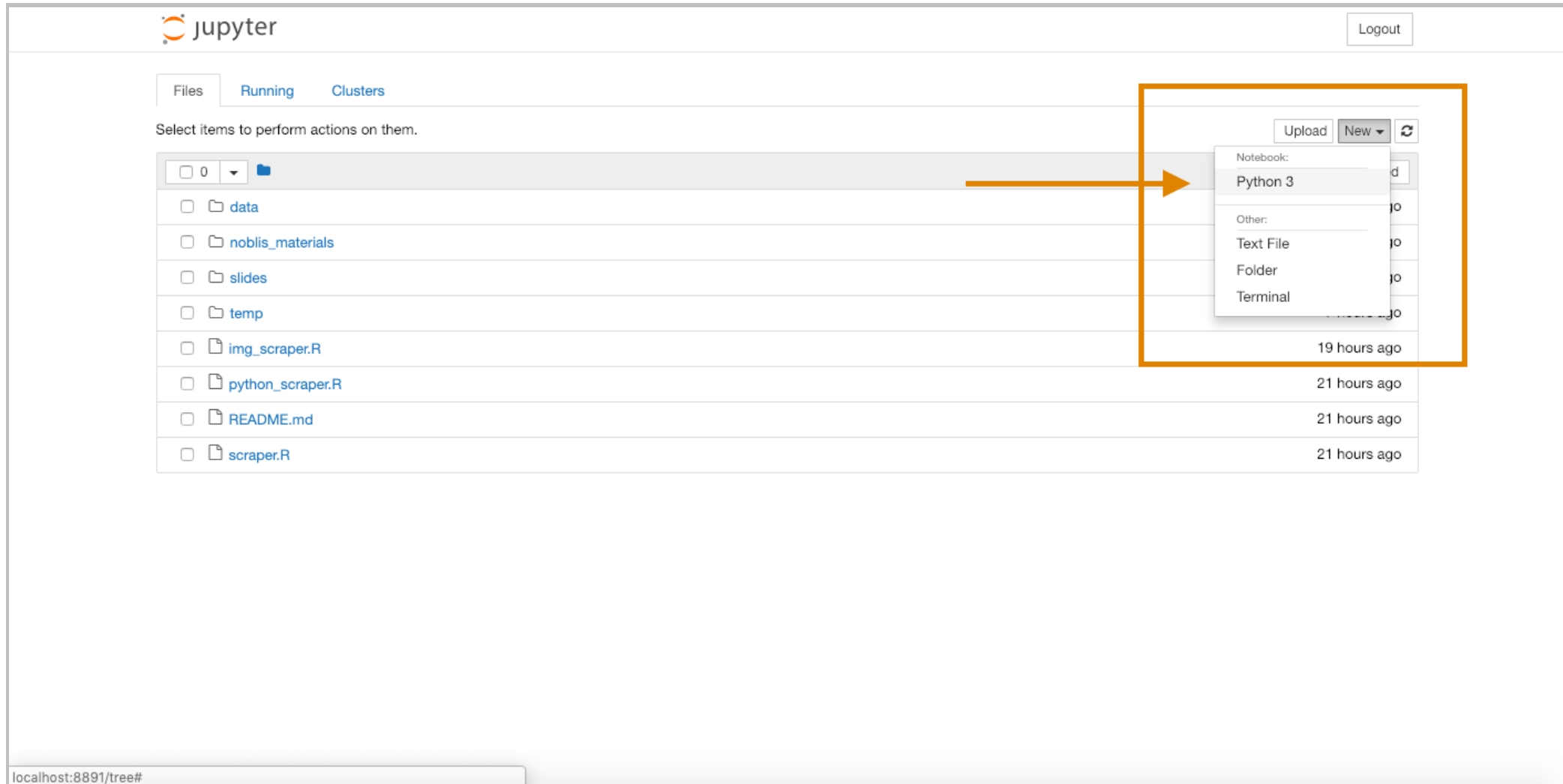
Launching Jupyter Notebook

- You can launch Jupyter simply from the [Anaconda launch screen](#)
- Click on **Jupyter notebook** and Anaconda will launch it for you



Using Jupyter Notebook

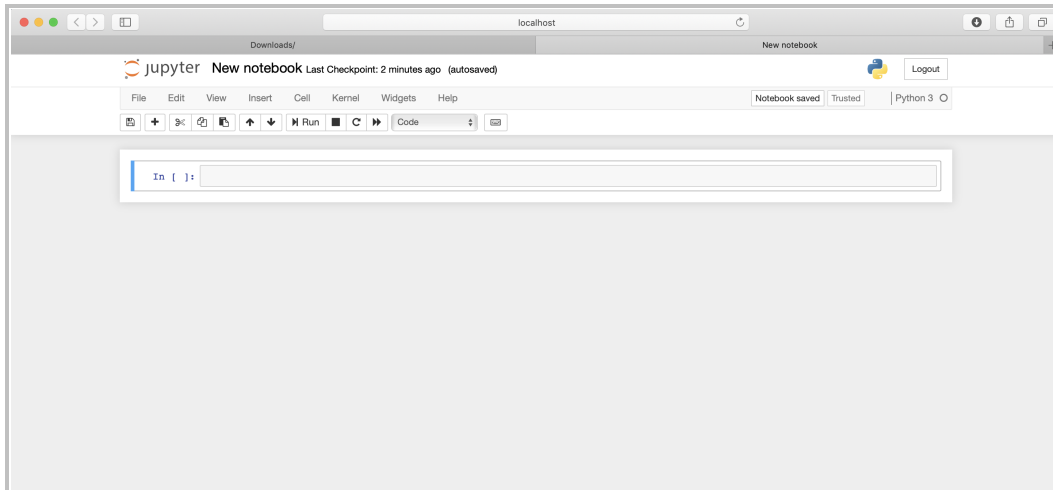
- To start a new script, click on 'New' at the top right hand corner and select **Python 3**



Using Jupyter Notebook

Output

- You should now see a new script with one line: `In [1]`
- This marks **Input 1** = `In [1]`



- You'll see the output on a new line labeled `Out [1]:` (and the number in brackets will increment each time)
- For the second command, be sure to include quotation marks around 'Hello students'
- You can use single or double quotes, just make sure they match!
- Let's run some commands in the console!

Running commands in Jupyter

- Try running each of these in the console one at a time

```
1 + 1
```

```
2
```

```
'Hello students'
```

```
'Hello students'
```

```
3 ** 2
```

```
9
```

```
10 * 3
```

```
30
```

```
4 / 3
```

```
1.3333333333333333
```

```
'a' + 'b'
```

```
'ab'
```

- Your notebook should look like this:

```
Running code

In [2]: 1 + 1
Out[2]: 2

In [3]: 'Hello students'
Out[3]: 'Hello students'

In [4]: 3 ** 2
Out[4]: 9

In [5]: 10 * 3
Out[5]: 30

In [6]: 4 / 3
Out[6]: 1.3333333333333333

In [7]: 'a' + 'b'
Out[7]: 'ab'
```

Essential keyboard shortcuts in Jupyter

- To find tricks and shortcuts, check under the menu at the top `Help > Keyboard Shortcuts`
- **We are going to review some of the most useful ones right now:**
- `Esc` will take you into “command mode” where you can navigate through cells using your keyboard

While in command mode:

- `A` to insert a cell above the current cell
 - `B` to insert a cell below the current cell
 - `M` to change the current cell to **Markdown**
 - `Y` to change it back to code
 - `D + D` have to press the key twice, deletes the current cell
- `Enter` will take you from command mode back into edit mode for the given cell
- `Ctrl + Enter` will run the current cell

Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	
Work with variables	
Summarize best practices for writing code	
Define various data types and perform basic operations	
Identify and create basic data structures	

Libraries and packages in Python

- **Module**: a `.py` file that defines one or more functions which you intend to reuse in different parts of your program

```
import <module> # to import the entire module  
from <module> import <classname> # imports a class from a module
```

- **Library**: this term is used loosely in Python, it is used to describe a collection of core modules. The *standard library* comes bundled with the core Python distribution

Set directories and install libraries

- We will start by setting our directories and installing the needed libraries for the day
- Good practice when writing code is to install all libraries you will need at the beginning of the script
- Today, we actually do not need to install any additional libraries to the standard library



Knowledge Check 1



Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	✓
Work with variables	
Summarize best practices for writing code	
Define various data types and perform basic operations	
Identify and create basic data structures	

Working with variables

- We can use variables to store values we wish to use again:

```
# Store the variables.  
x = 5  
name = 'Mike'
```

```
# Print them out.  
print(x)
```

5

```
print(name)
```

Mike

Defining multiple variables at once

- We can also define several variables as follows:

```
a, b, c = 1, 2, 3
```

- Now try and print each individual variable

Swapping variables

- Python also gives us the ability to swap variables like so:

```
a, b = b, a
```

- Now try printing both `a` and `b`, what do you see?

Printing and deleting variables

- Print the x variable:

```
print('x = ', x)
```

```
x = 5
```

```
print('x + 1 = ', x + 1)
```

```
x + 1 = 6
```

```
print('x * x = ', x * x)
```

```
x * x = 25
```

- You can delete a variable with the `del` command:

```
del x
```

- Now `x` will be gone from your variable explorer, and you won't be able to access its value

Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	✓
Work with variables	✓
Summarize best practices for writing code	
Define various data types and perform basic operations	
Identify and create basic data structures	

Questions to keep in mind while writing code

- What do we do with **runtime errors**?
- How do we annotate code and **make code reproducible**?



Runtime errors

- What happens if our code hits an error?
- Test this by adding the line `print (y)` towards the end of your script
- We haven't defined a variable `y`, so this will cause an error
- Run this line now, and you will get an error:

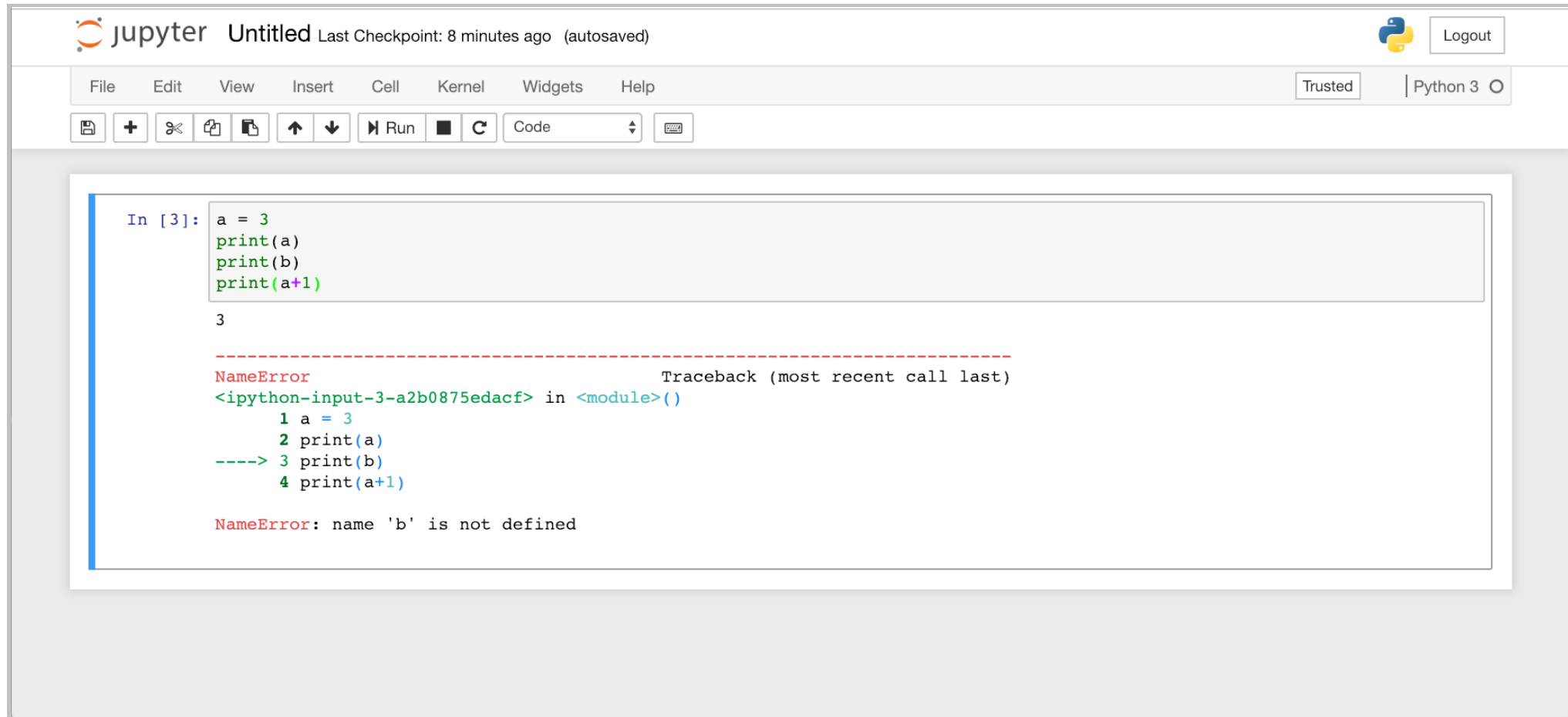
```
NameError: name 'y' is not defined
```

Runtime errors - what do we do?

- Go back to our code
- Find the error
- Read the messages related to the error
AND debug!



Runtime errors - example



The image shows a Jupyter Notebook interface. At the top, the header bar includes the Jupyter logo, the text "jupyter Untitled", and "Last Checkpoint: 8 minutes ago (autosaved)". On the right, there is a "Logout" button and a Python logo. Below the header is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". To the right of the menu bar are "Trusted" and "Python 3" buttons. Below the menu bar is a toolbar with icons for saving, adding, deleting, and running cells, as well as a "Code" dropdown menu. The main area of the notebook contains a code cell with the following text:

```
In [3]: a = 3
print(a)
print(b)
print(a+1)
```

The output of the cell shows the number "3" followed by a red dashed line and a traceback:

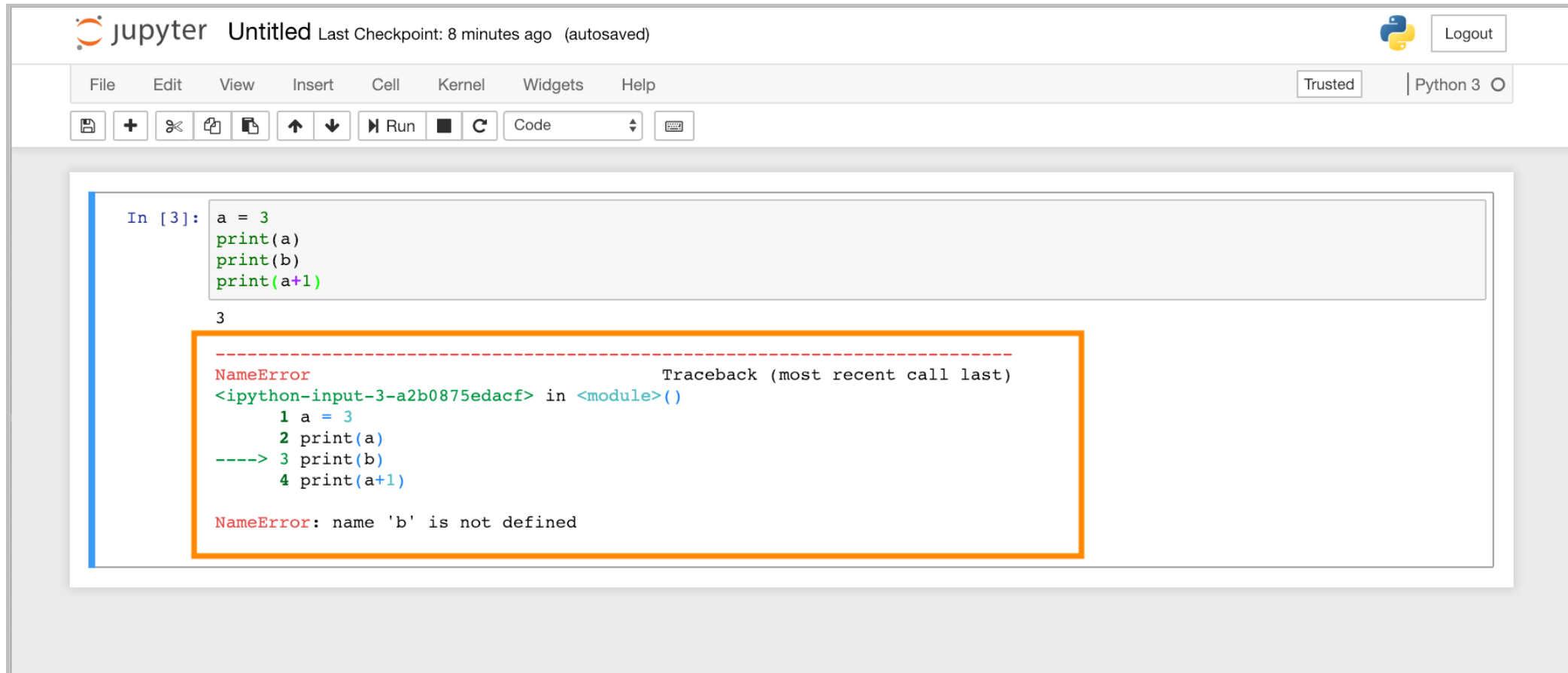
```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-a2b0875edacf> in <module>()
      1 a = 3
      2 print(a)
----> 3 print(b)
      4 print(a+1)

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

Runtime errors - step 1

Go back to our code

- We know where the error occurs because the code will stop running at that specific chunk in our notebook



The screenshot shows a Jupyter Notebook window titled "Untitled" with a last checkpoint 8 minutes ago. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, undo, redo, and running code. The code cell contains the following Python code:

```
In [3]: a = 3
print(a)
print(b)
print(a+1)
```

The output of the code cell shows the number 3, followed by a traceback for a `NameError`. The traceback is highlighted with an orange border and shows the following details:

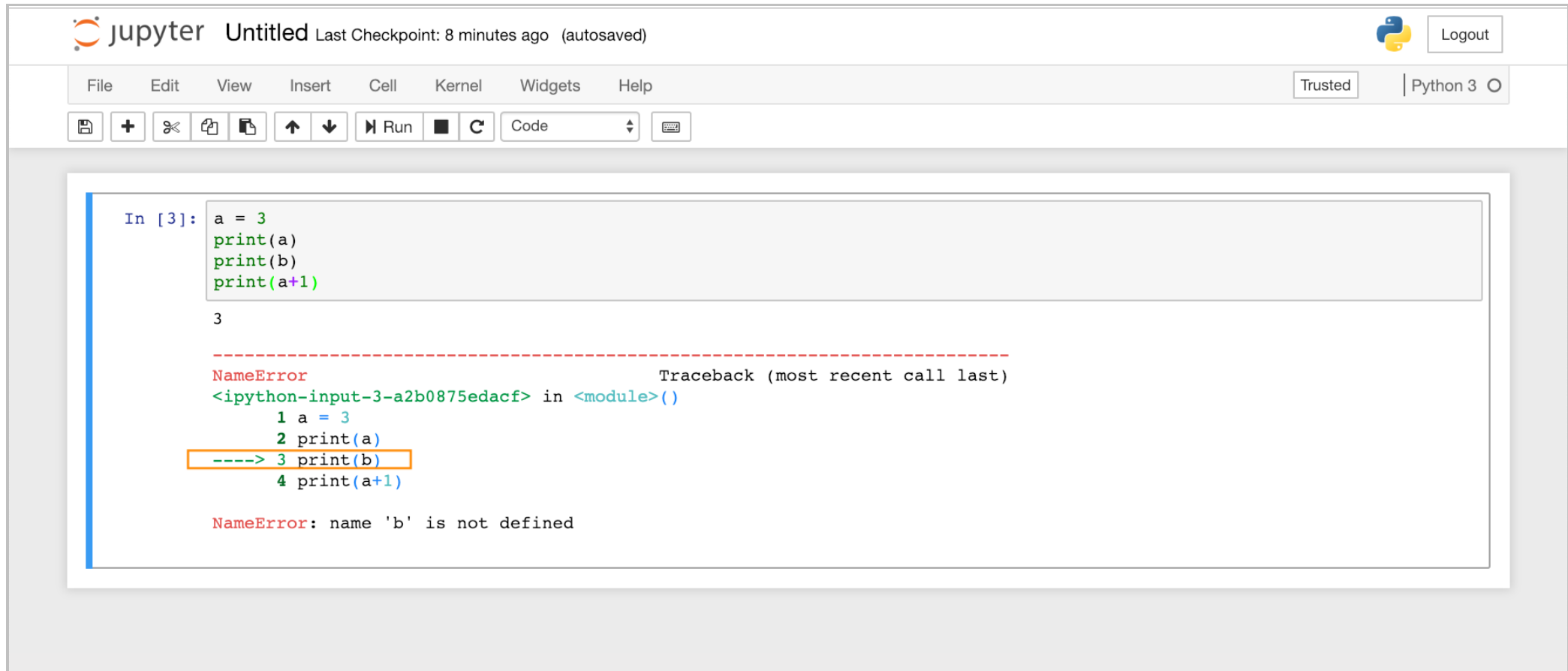
```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-a2b0875edacf> in <module>()
      1 a = 3
      2 print(a)
----> 3 print(b)
      4 print(a+1)

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

Runtime errors - step 2

Find the error

- If there are multiple lines in the specified code chunk, Jupyter will point out the particular line where the error occurred



The screenshot shows a Jupyter Notebook window titled "Untitled" with a last checkpoint of 8 minutes ago. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, undo, redo, and running code. The code cell contains the following Python code:

```
In [3]: a = 3
        print(a)
        print(b)
        print(a+1)
```

The output of the code cell shows the number 3, followed by a red dashed line indicating a traceback. The traceback shows the error occurred in the third line of the code cell:

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-a2b0875edacf> in <module>()
      1 a = 3
      2 print(a)
----> 3 print(b)
      4 print(a+1)

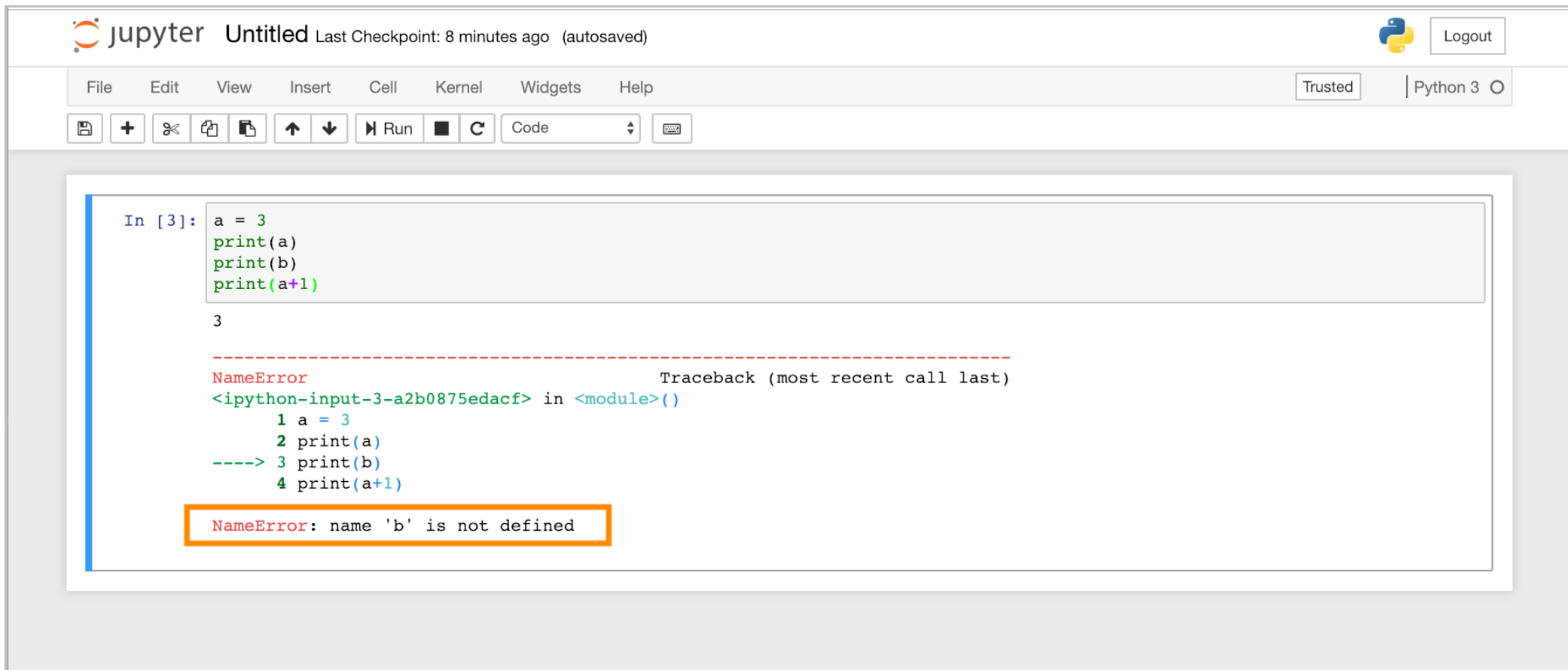
NameError: name 'b' is not defined
```

The line `print(b)` is highlighted with an orange box, indicating the line where the error occurred.

Runtime errors - step 3

Read the messages related to the error AND debug!

- We then look at the error type: “NameError” in this case
- Python has many different types of errors



The screenshot shows a Jupyter Notebook window titled "Untitled" with a "Last Checkpoint: 8 minutes ago (autosaved)" status. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, undo, redo, and running code. The code cell contains the following Python code:

```
In [3]: a = 3
        print(a)
        print(b)
        print(a+1)
```

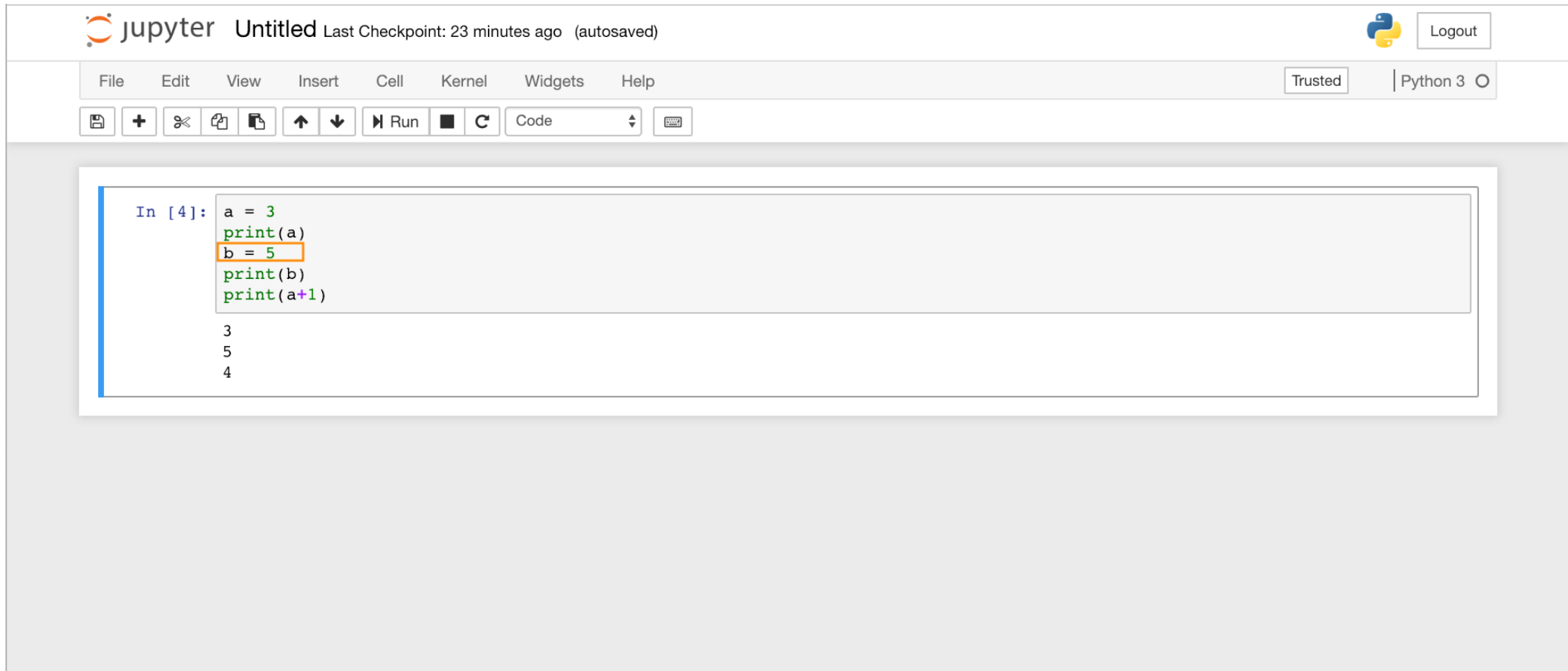
The output of the code is the number 3. Below the output, a traceback is displayed, indicating a `NameError` occurred. The traceback shows the following code lines:

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-3-a2b0875edacf> in <module>()
      1 a = 3
      2 print(a)
----> 3 print(b)
      4 print(a+1)
```

The error message at the bottom of the traceback is `NameError: name 'b' is not defined`, which is highlighted with an orange box.

Runtime errors - finally error free!

- The error was a `NameError`
- We defined `b`
- The error is now fixed and the code runs!



The image shows a Jupyter Notebook interface. At the top, the header bar displays the Jupyter logo, the text "jupyter Untitled", and "Last Checkpoint: 23 minutes ago (autosaved)". On the right side of the header, there is a Python logo and a "Logout" button. Below the header is a menu bar with options: File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. To the right of the menu bar, there is a "Trusted" status indicator and a "Python 3" dropdown menu. Below the menu bar is a toolbar with icons for saving, adding a new cell, undo, redo, copy, paste, and other standard editing functions. The main area of the notebook contains a code cell. The code in the cell is as follows:

```
In [4]: a = 3
        print(a)
        b = 5
        print(b)
        print(a+1)
```

The output of the code cell is displayed below the code:

```
3
5
4
```

Annotating your code

- Reproducibility of your code is a key part of being a productive data scientist or programmer
- Always remember to **annotate your code** and **explain what you are doing**
- **The goal is to annotate your code so that any other data science team member can execute your script correctly and understand what is happening, line by line**



Commenting your code - line by line

- We want others to be able to walk through our code and understand what we are doing without having to talk to the actual author of the code
- In Python, you write comments by starting the line with a '#' sign

```
# In this step, we add the variables a and b to create a new variable d.  
d = a + b  
# Now we print d.  
print(d)
```

3

Commenting your code - by chunk

- Sometimes, you may want to comment about a chunk of code, maybe a function that you do not want to comment out line by line
- You can comment multiple lines without using the #
- You can use three quotation marks before and after the commenting area:

```
"""  
In this step, we read in the data from the flat file. Each row corresponds  
to a different subcounty. We break the long geoid into separate IDs for  
state, county, and subcounty.  
"""
```

Knowledge Check 2



Exercise 1



Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	✓
Work with variables	✓
Summarize best practices for writing code	✓
Define various data types and perform basic operations	
Identify and create basic data structures	

Data types

- Now that we've had some practice working with variables, it's time to understand the way Python categorizes different values
- **The three main data types are:**
 - Numbers
 - Logicals
 - Strings



Numbers: integer and floating point

- Numbers are just what they sound like
- 7, -1, and 4.312897 are all numbers
 - *Floating point numbers*: numbers that have decimal values
 - *Integers*: numbers without any decimal values

Numbers: figuring out the type

- Throughout this section, we will be using Python's `type` function, which gives us the data type of a particular value
- Try running the following:

```
type(1.5)
```

```
<class 'float'>
```

```
type(7)
```

```
<class 'int'>
```


Numbers: practice

- Now guess the output for the statements below, and then run each line to see if you were right

```
type(-1)
type(2.32498)
type(1 + 3.5)
type(1.5 + 3.5)
type(4/2)
```

Numbers: practice - output

```
print(type(-1))
```

```
<class 'int'>
```

```
print(type(2.32498))
```

```
<class 'float'>
```

```
print(type(1 + 3.5))
```

```
<class 'float'>
```

```
print(type(1.5 + 3.5))
```

```
<class 'float'>
```

```
print(type(4/2))
```

```
<class 'float'>
```

Rounding

- Python has three types of rounding

Rounding method	Definition	Example
round	If the decimal value is 0.5 or higher, round up, otherwise round down	<code>round(9.1)</code>
floor	Rounds down no matter what	<code>floor(9.9)</code>
ceil	Rounds up no matter what	<code>ceil(9.1)</code>

- To use `floor` and `ceil`, you have to import them from the `math` library
- You will have the `math` library because *Anaconda has it pre-installed*

Booleans

- Logical values, also called Booleans, can take one of two values:
 - True
 - False
- To create a Boolean variable, you can simply declare it as follows:

```
i_like_cake = True  
i_like_broccoli = False  
print(type(i_like_cake)) # tells you the type of this variable is "bool"
```

```
<class 'bool'>
```

Booleans

- You can get the opposite value of a Boolean variable using the `not` keyword:

```
print(not i_like_cake)
```

```
False
```

```
print(not i_like_broccoli)
```

```
True
```

- **Best practice:** because Booleans have a `True`/`False` value, it is good practice to use variable names that sound like a question, such as `i_like_cake`, or `is_ready`, `has_name`, etc
- This helps make your code easier to understand when someone else is reading it

Arithmetic with Booleans

- At times, Booleans can act as numbers!
- Python will convert (“implicitly cast”) any `True` values into 1, and `False` values into 0
- For example:

```
print(i_like_cake + i_like_broccoli)
```

```
1
```

```
print((not i_like_cake) + i_like_broccoli)
```

```
0
```

Logical operators

- The most common way to create Booleans is by using *logical operators*
- When you compare numbers or variables using these operators, your return will be a Boolean value ('True' or 'False')

Operator	Example
Greater than	<code>x > y</code>
Less than	<code>x < y</code>
Equal to	<code>x == y</code>
Not equal to	<code>x != y</code>
Greater than or equal to	<code>x >= y</code>
Less than or equal to	<code>x <= y</code>

Logical operators in action

- Let's look at the operators in action

```
# First, we set two variables to equal 7 and 2 respectively.
```

```
x = 7  
y = 2
```

```
# Now we see if x is greater than y.  
print(x > y)
```

True

```
# Let's check if y is smaller than x.  
print(y < x)
```

True

```
# We check for greater than and less than in a similar manner.  
print(y <= x)
```

True

```
# And if x is equal to or not equal to y.  
print(x == y)
```

False

```
print(x != y)
```

True

Combining Booleans

- We can use `and` and `or` to check a combination of conditions
- Let's say we want to check if `x` is larger than 5 but less than 10

```
# Then we can ask:  
print(x > 5 and x < 10)
```

```
True
```

- We can add as many statements as we want with `and`: the interpretation being to check all of the conditions, and only return a `True` if **all** of the conditions are met

```
# For example:  
print(x > 5 and x > 1 and abs(x) == 7 and x < 10)
```

```
True
```

```
# And when all the conditions are not true, even though one is x < 10:  
print(x >= 8 and x < 10)
```

```
False
```

Strings

- Think of the word “strings” as being synonymous with “text”
 - The value `"1600 Pennsylvania Ave NW"` is a string
 - So is `"a"`, `"00123"`, and even `"1"`
- By surrounding a value with quotation marks, Python treats it as a string
- Either single quotes (`'`) or double quotes (`"`) can be used to create a string



Functionality of common string functions

- To learn the functionality of some common string functions, we first will create two strings to work with

```
first_name = 'lisa'  
last_name = 'smith'
```

- `len` is used to find the number of letters in each string

```
print(len(first_name))
```

```
4
```

- `+` will combine two strings

```
full_name = first_name + last_name  
print(full_name)
```

```
lisasmith
```

- Make sure to add a ' ' space!

```
full_name = first_name + ' ' + last_name  
print(full_name)
```

```
lisa smith
```

Functionality of common string functions

- Remember, although we are using the + operator, it will always concatenate and not add numbers if they are written as strings

```
print("1" + "2")
```

```
12
```

- What happens if we try and concatenate a string and a number? Try it. "1" + 2
- We get a type error because we must have the same data type to add two variables together
- We can solve this by casting 2 as a string

```
print("1" + str(2))
```

```
12
```

A few more common string functions

Function	Definition	Example
<code>.lower()</code>	Convert string to all lowercase	<code>full_name.lower()</code>
<code>.upper()</code>	Convert string to all uppercase	<code>full_name.upper()</code>
<code>.capitalize()</code>	Capitalizes the first letter of the string	<code>full_name.capitalize()</code>
<code>.find()</code>	Replace every occurrence of a letter or group of letters with something else	<code>full_name.find('lisa')</code>
<code>.replace()</code>	Convert string to all uppercase	<code>full_name.replace('lisa','sara')</code>
<code>.strip()</code>	Strip extra whitespace from text	<code>full_name.strip()</code>

String slicing

- To take a substring from a string, we can use Python's string slicing notation
- To get a single character from a list, specify a single number

```
letters = 'ABCDEFGH'  
print(letters[2])
```

C

- To get a range of characters, specify the start and endpoint separated by a colon
- The “end” value should be one larger than the end value you want

```
print(letters[2:5])
```

CDE

- You can specify a position counting from the back by using a negative number

```
print(letters[-1])
```

H

- A shorthand that is built off of this is a quick way to reverse the order of a list

```
print(letters[::-1])
```

HGFEDCBA

- Remember, the index starts at 0!

Changing a variables data type

- In Python, a variable is defined based on the type that it is interpreted as

```
name = "Bob" # no need to say that name is a
string
name = "Ann" # overwriting with another string
is fine
print(type(name))
```

```
<class 'str'>
```

```
name = 15 # in fact, overwriting with anything
at all is fine!
print(type(name))
```

```
<class 'int'>
```

- Suppose we want to convert a variable which is a float into an integer

```
# For example:
x = 5.8
print(type(x)) # it's a float
```

```
<class 'float'>
```

```
int(x) # turns x into an integer, keeping just
the integer part
```

```
5
```

```
print(int(x))
```

```
5
```

Knowledge Check 3



Exercise 2



Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	✓
Work with variables	✓
Summarize best practices for writing code	✓
Define various data types and perform basic operations	✓
Identify and create basic data structures	

Data structures

- Now that we've covered the base types: numbers (floats & ints), strings, and Booleans, we will look at some composite types
- Composite types combine base types into new structures
- Composite types are useful when we want to create a variable that contains multiple values
- The four major composite types we'll look at are:
 - Lists
 - Sets
 - Dictionaries
 - Tuples



Lists

- The most simple data structure is a `list`
- A list is just what it sounds like, it contains a collection of values
- The elements of a list can be of any data type, as we see with `mixed`
- List members can even be other lists, as shown with `addresses`, and `mixed`
- A list can have the same value in it multiple times, as shown with `ages`
- A list can exist but have no elements, as shown with `empty_list`

```
int_list = [1,2,3,4] # contains four integers
str_list = ['texas', 'delaware', 'california'] #
contains three strings
addr1 = [1600, "Pennsylvania Ave NW",
"Washington", "DC", 20500]
addr2 = [1213, "U St NW", "Washington", "DC",
20009]
addresses = [ addr1, addr2 ]
mixed = [1, True, 3.4, 'Hello', addr1]
ages = [ 10, 12, 11, 12, 10, 11, 12]
empty_list = []
```

Accessing members of a list

- When we define a list, we declare the members in a particular order
- This ordering lets us pull out values from the list
- Each member of the list can be retrieved using the number corresponding to its position in the list
- Python begins counting with 0
- So the first member of a list is said to be in “position 0,” the second member is in “position 1”, and so on

```
addr = [1600, "Pennsylvania Ave NW",  
        "Washington", "DC", 20500]  
street_num = addr[0]  
street = addr[1]  
city = addr[2]  
state = addr[3]  
zipcode = addr[4]
```

Max, min and sum

- If our list contains only numbers, or only strings, we can take the max and min values:

```
num_list = [100, 115, 112, 98]
print(max(num_list))
```

115

```
print(min(num_list))
```

98

```
# A list of strings.
str_list = ['Philip', 'Louis', 'Margaret']
max(str_list)
```

'Philip'

- If our list contains only numbers, we can take the sum of all the values:

```
num_list = [100, 115, 112, 98]
print(sum(num_list))
```

425

Split and join

- Two very useful string methods allow us to convert between strings and lists:
 - `.split()`
 - `.join()`
- Suppose we have a long string and want to split it using a comma

```
addr = '1600 Pennsylvania Ave NW, Washington DC, 20500'  
  
# Split address by comma.  
addr_parts = addr.split(',')  
print(addr_parts)
```

```
['1600 Pennsylvania Ave NW', ' Washington DC', ' 20500']
```

```
# Join addr back together  
addr_orig = ",".join(addr_parts)  
print(addr_orig)
```

```
1600 Pennsylvania Ave NW, Washington DC, 20500
```

Append - add an item to list

- Two very useful methods to add and remove items from a list are:
 - `.append()`
 - `.pop()`
- To add elements to the end of a list, we use the `.append` method

```
names = ['Beth', 'Sara', 'Tracey', 'Thomas']  
print(len(names)) # we have 4 names
```

```
4
```

```
names.append('Christopher')  
print(names)
```

```
['Beth', 'Sara', 'Tracey', 'Thomas', 'Christopher']
```

```
print(len(names)) # now we have 5 names
```

```
5
```


Pop - remove item from list

- To remove a particular list member, use the `.pop()` method:

```
deleted_name = names.pop()  
print(names)
```

```
['Beth', 'Sara', 'Tracey', 'Thomas']
```

```
print(deleted_name)
```

```
Christopher
```

```
print(len(names)) # now we're back to 4 names
```

```
4
```

- Note the value that was removed is the output as a result of the `.pop()` call

Index

- What if we want to know where to find a particular value in a list?
- We have a list of names and want to know which position the name 'Thomas' is:

```
names = ['Michael', 'Abigail', 'Thomas', 'Brianna']  
  
# We use .index to find the name.  
thomas_idx = names.index('Thomas')  
print(thomas_idx)
```

2

- Remember, list positions start counting from 0!

Tuples

- Tuples are similar to Python lists in that they allow elements of any data type
- Though you typically will not create them outright, they appear in different areas of Python
- A tuple is denoted with parentheses:

```
tup = (1, 'a', [5, 6, 7])
```

- A very commonly used feature of tuples is that their values can be extracted into variables

```
num, char, lst = tup  
print(num)
```

```
1
```

```
print(char)
```

```
a
```

```
print(tup)
```

```
(1, 'a', [5, 6, 7])
```

Sets

- A set is like a list that does not allow duplicate values
- To create a set, we use curly braces, { }, instead of square brackets []

```
print({1, 10.5, -6})
```

```
{1, 10.5, -6}
```

```
print({'DE', 'MA', 'WI', 'TX'})
```

```
{'DE', 'TX', 'WI', 'MA'}
```

```
print({5, 'maryland', 8/12})
```

```
{0.6666666666666666, 'maryland', 5}
```

- What else do you notice happening in each set?

Converting a list into a set

- We can also convert a list into a set, by using the `set ()` function:

```
mylist = [1,1,1,20]
print(set(mylist))
```

```
{1, 20}
```

- Notice how only unique values are kept
- Note that, unlike lists, sets are considered “unordered”
- With a list, if we wanted to retrieve the first value, we could say:

```
mylist = [3, 7, 10, 1.5]
val1 = mylist[0]
```

- This is not supported with sets
- For this reason, we will **mostly be using lists and dictionaries (which is covered next) to store data**
- Sets are useful for obtaining unique values of a list

Dictionaries

- With **lists**, we have to know the position of the item to fetch it
- With **dictionaries**, we use **key-value pairs**
- Let's build a dictionary with an address

```
addr = {"street_num": 1600, "street": "Pennsylvania Ave NW", "city": "Washington",  
        "state": "DC", "zipcode": 20500}
```

Let's break down the syntax above:

1. Curly braces are required to make a dictionary
2. We also must provide key-value pairs, the syntax is ``key: value``
3. Multiple key-value pairs are separated by commas

Print the dictionary

- Print the dictionary to the console:

```
print(addr)
```

```
{'street_num': 1600, 'street': 'Pennsylvania Ave NW', 'city': 'Washington', 'state': 'DC', 'zipcode': 20500}
```

- To extract a member of a dictionary, we refer to the name of the key we want
- For example, if we want the zip code, we reference that key `print(addr['zipcode'])`

- If we try to access a key that doesn't exist, Python will throw a `KeyError`:
`addr['country'] # ==> KeyError: 'country'`
- Remember to not treat dictionaries like lists. The following will not work, again throwing a `KeyError`: `addr[0] # ==> KeyError: 0`

Creating a dictionary with zip

- When we have two lists, one with the keys and another with the values, we can use the `zip` function to create a dictionary
- For example:

```
keys = ['street_num', 'street', 'city', 'state', 'zipcode']  
values = [1600, 'Pennsylvania Ave', 'Washington', 'DC', 20500]
```

- Then, we can say:

```
addr = dict(zip(keys, values))  
print(addr)
```

```
{'street_num': 1600, 'street': 'Pennsylvania Ave', 'city': 'Washington', 'state': 'DC', 'zipcode': 20500}
```

- The output of `zip` is something like a list of tuples, and `dict` converts this to a dictionary

Other functions to use with a dictionary

Function	Definition	Example
<code>del</code>	Delete a member of a dictionary referencing its key	<code>del addr['zipcode']</code>
<code>.keys()</code>	See the keys in the dictionary	<code>addr.keys()</code>
<code>.values()</code>	See the values in the dictionary	<code>addr.values()</code>
<code>.copy()</code>	Duplicate a dictionary	<code>addr2 = addr1.copy()</code>
<code>.update()</code>	Merge two dictionaries	<code>addr1.update(addr2)</code>

Knowledge Check 4



Exercise 3



Module completion checklist

Objective	Complete
Summarize data science use cases for Python vs. R	✓
Use Anaconda and Jupyter Notebook to access and write code	✓
Install all packages that will be needed for the course	✓
Work with variables	✓
Summarize best practices for writing code	✓
Define various data types and perform basic operations	✓
Identify and create basic data structures	✓

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today

Today, you will:

- Define variables of different types
- Manipulate variables of different types
- Combine variables into basic data structures
- Manipulate basic data structures

This completes our module
Congratulations!