Intro to Python III

ASTR 2910 ⋆ Week 6

Recap of last time(s)

Part I

- **Variables:** Storing information for later use; multiple **data types**
- **Boolean operators:** Making comparisons between quantities
- **Conditionals:** Branching decisions (based on Boolean expressions)
- **Loops:** Repetitive actions (over an iterable or until a condition is met)

Part II

- **Functions:** Generalized and modular to make reusing code easy!
- **Packages:** Code written by others that you can import and use

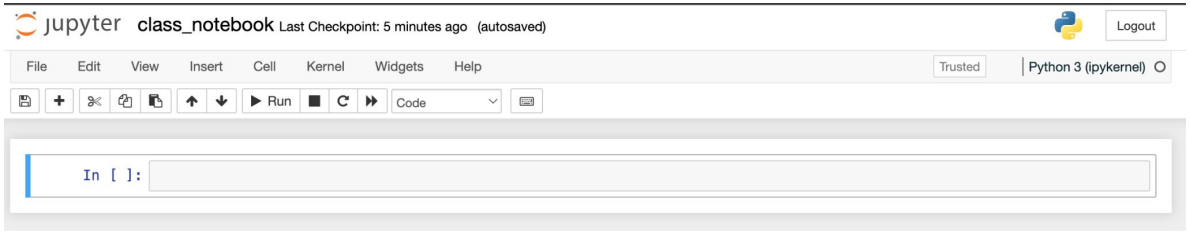
Jupyter Notebook

Launching Jupyter Notebook

1. For Mac/Linux: Terminal > `conda activate`
   For Windows: Start Menu > Anaconda Prompt
   In both cases, you should see (base) at the beginning of your prompt.

2. `jupyter notebook`
   If this command doesn't work: `pip install jupyter-notebook`
   Jupyter Notebook should automatically launch in a new browser tab.

3. Navigate to where you want your notebook > New > Python 3



What is a Jupyter Notebook?

When conducting scientific research, it's often useful to be able to intermingle text, figures, and code. Jupyter Notebooks are a Python file format that allow this.

When you open a notebook, a Python **kernel** starts in the background – like the iPython interpreter we used in the terminal, but better.

You can then write code in chunks (called "cells") that can be run in any order.

Cells can also contain text (change the cell type to "markdown").

Benefits and caveats

Benefits:

1. Much easier to make plots, which show up in-line with your code
2. Can easily run and re-run individual chunks of code
3. Easier to share code with others

Caveats:

1. Not suitable for writing installable packages/software
2. Because the kernel stays active the entire time, old variables stay defined, which can mess up your results if you forget to change a name somewhere or run cells out of order

Classes

Introduction to object-oriented programming

Technically, everything in Python is an **object**.

Objects are created from **classes**, which are generalized pieces of code that specify how an object behaves. (Think of classes like data types.)

An object is a specific **instance** of a class. For example: all strings have the same properties, but each string that you create is a unique object.

Object-oriented programming is the practice of using user-defined classes and objects to streamline your code.

Attributes and methods

Classes (and therefore objects) have two main kinds of properties:

1. Attributes: Discrete pieces of information, like variables
2. Methods: Functions that operate on the objects belonging to that class

Example with a `numpy` array:

```
arr = np.array([[1,2,3,4], [5,6,7,8]])

#Attributes
arr.size
arr.shape

#Methods
arr.sum()
arr.flatten()
```

Both types of properties are accessed with "dot" notation: `variable_name.property`

Methods are functions, hence the ().

Attributes have the same name, but unique values, for each object.

Defining your own classes

To define your own classes, you must use a specific syntax:

```
class Rectangle:
    def __init__(self, left_upper: tuple, right_lower: tuple):
        self.left_upper = left_upper
        self.right_lower = right_lower
        self.width = right_lower[0]-left_upper[0]
        self.height = right_lower[1]-left_upper[1]

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return self.width * 2 + self.height * 2
```

See Chapter 10 in your textbook for more details! We won't be exploring classes in detail this semester.

I/O

Reading basic text files

To read text files, we can use the built-in `open()` function. The first argument will be the file path, and the second argument will be the opening mode (`'r'` = read).

Syntax option 1:

```
file = open('example_data.txt', 'r')
#do something with the file here
file.close()
```

Syntax option 2:

```
with open('example_data.txt', 'r') as file:
    #do something with the file here
    pass
```

In both cases, the file is being closed! The difference is that option 2 will close the file automatically when the `with` statement is exited.

Reading basic text files

Once you have the file stored in a variable, you can read the data in many ways.

Read all lines into a list:

```
lines = file.readlines()
```

Read line by line:

```
for line in file:
    pass
```

Each line will be stored as a string, which you can manipulate using functions like `.split()` and casting to pull out the data you want.

You can also use `line.startswith('#')` and `continue` to skip comments.

Writing basic text files

To write your own file, use the `open()` function and specify the 'w' mode:

```
with open('example_data.txt', 'w') as file:
    file.write('# star id\tmagnitude\n')
    for i, v in enumerate(random_mags):
        file.write(f'{i}\t{v:.2f}\n')
```

You must add your own line breaks!

BEWARE: if you feed in the name of an existing file, it will be wiped + overwritten!

A file will be created for you at the path you provide if one doesn't already exist.

Try to write your file in a way that will be easy to read back in.

Working with CSVs

CSV files have a common structure: each newline represents a row of data, and the values are separated only by commas.

Several packages have dedicated ways to read in `.csv` files:

```
npt = np.genfromtxt('example_data.csv', delimiter=',')
```

```
from astropy.table import Table
t = Table.read('example_data.csv')
```

```
import pandas as pd
pdt = pd.read_csv('example_data.csv')
```

Other file types

Many Python packages have functions that can help you read in different types of files: Excel spreadsheets, `.npy` files, etc.

When in doubt, check the documentation!

`astropy.Table()` in particular has a lot of parameters that can be tweaked to adapt to tricky text files, but also check out `pandas` and other `numpy` functions.