Introduction to Python

Chris Cornwell

Dec 17, 2024

Working in Python throughout the semester.

Often interact with Python through Jupyter notebooks (`.ipynb` files – *IPython notebook*).

Two approaches to run and work with Jupyter notebooks.

1. Google's Colaboratory – *easiest startup*, done in the cloud, extra effort to interact with other files.
2. Install and run Python and Jupyter on your computer – *installation steps to get started*, can work offline, easy to interact with other files.

Instructions for the two approaches.

Jupyter notebooks in a nutshell: Markdown cells and Code cells.

- Markdown cells: use HTML code; a lot of syntax shortcuts for formatting & styling. A guide for writing in Markdown.
- Code cells: write lines of Python code. The notebook has a Python kernel (session) running; when you “Run” or execute a Code cell, the notebook works in that Python session and displays the output (if any).

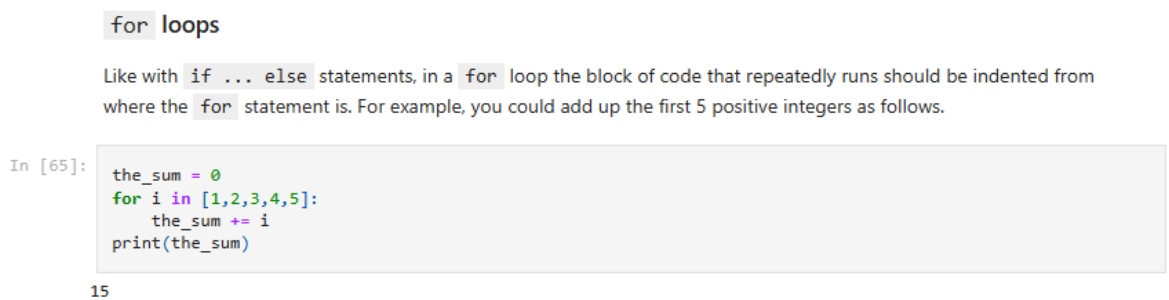

Figure: A Markdown cell followed by a Code cell in a Jupyter notebook

Assigning variables

- A variable is assigned by placing, on one line, `<variable name> = <assigned value>`.

```
1 | x = 5.11
2 | y = 5
3 | name_full = 'Chris Cornwell'
```

↑ this code assigns three variables, `x`, `y`, and `name_full`

- To “comment out” a line, begin line with #. Good for notes to yourself, or others reading the code.

```
1 | # Make an ordered pair; output would be (10.11, 4)
2 | (x + y, y - 1)
```

- Possible to assign more than one variable in one line.

```
1 | x, y = 5.11, 5
```

Data type

- Each variable has a *data type* (or, simply *type*). In

```
1 x = 5.11
2 y = 5
3 name_full = 'Chris Cornwell'
```

the types of the assigned vars are **float**, **int**, and **str** respectively.
Unlike in other programming languages, you don't have to declare the types of the variables. Python *interprets* it. (And it can change at some later point in the flow of the code.)

- type **int**: like an integer.
- type **float**: like a real number in decimal form . . . *kind of*.
- type **str**: a "string," or sequence of *characters* (that can be typed from keyboard). Will return to this again.

Numerical types

The four main operations[1] `+`, `-`, `*`, and `/` work as you would expect on numerical types **int** and **float**. Unlike when writing math, you cannot leave out `*` when multiplying.

Question

Why would it be a *bad* idea to have Python interpret something like **ab** as being "**a** times **b**"?

Assigning after an operation. **Very** often want to change a variable by some amount (e.g., increase it by 1); have it keep new value.

```
1 | y = y + 1
2 | # Line above has convenient shorthand, below
3 | y += 1
```

This is *not* a mathematical equation, but an assignment. The shorthand works for other operations.

[1] Representing addition, subtraction, multiplication, and division.

Logical types and `None`

We have some logical types, as every language needs – True and False.
Usually, no need to directly assign or work with these. They are "under the hood" when making comparisons.

- Technically, True and False are like `1` and `0` in Python. Use this fact only with *extreme care*! (Maybe just avoid trying to use it.)

```
1 | True*False
```

  ↑ the line above will return `0`.

The *null* type in Python is `None`. We'll talk about using it in later lectures.

Basics of lists

- **`list`** is a data *sequential* type in Python – it holds a sequence of "items." Each could be an **`int`**, each could be a **`list`**, or there could be some **`int`** type items and some **`str`** type.
- Assigning a **`list`** variable:

```
1 | my_list = [2, 3, 5, 'p']
2 | empty_list = []
```

To refer (and get access to) a list item use its index, starting at 0: `my_list[0]` is 2 above, `my_list[1]` is 3, and so on.
The + operation is defined on strings. It results in the *concatenation* of the lists – putting them together, end to end.

```
1 | # the code below outputs [2, 3, 5, 'p', 11, 13]
2 | my_list + [11, 13]
```