Introduction to Python

Chris Cornwell

Dec 17, 2024

Outline

Outline

Working in Python throughout the semester.

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

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.

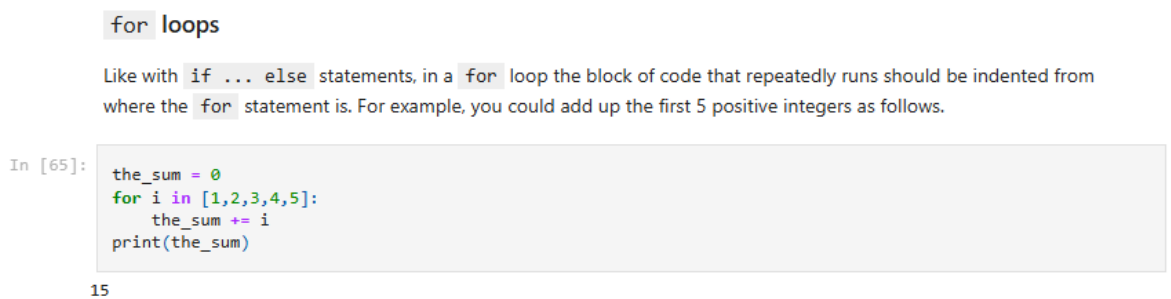
- Markdown cells, for text: can use HTML code; syntax shortcuts for formatting & styling. A guide for writing in Markdown.

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

- Markdown cells, for text: can use HTML code; 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).

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

- Markdown cells, for text: can use HTML code; 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).



`for` loops

Like with `if ... else` statements, in a `for` loop the block of code that repeatedly runs should be indented from where the `for` statement is. For example, you could add up the first 5 positive integers as follows.

In [65]:

```
the_sum = 0
for i in [1,2,3,4,5]:
    the_sum += i
print(the_sum)
```

```
15
```

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

Outline

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`

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)
```

Assigning variables

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

```
x = 5.11
y = 5
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.

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

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

```
x, y = 5.11, 5
# or, you could use
x = 5.11; y = 5
```

Data type

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

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

↑ the types of the assigned vars are **`float`**, **`int`**, and **`str`** respectively.

Data type

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

```
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, don't need to declare the types of the variables. Python *interprets* it. (The type might even *change* at a later point.)

- 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.

Outline

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.

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

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`"?

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

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`); to keep new value, *reassign* after the operation.

```
y = y + 1
# A convenient shorthand for line above is
y += 1
```

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

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`); to keep new value, *reassign* after the operation.

```
y = y + 1
# A convenient shorthand for line above is
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.

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 it.)

```
True*False
```

↑ the line above will return `0`.

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 it.)

```
True*False
```

↑ the line above will return `0`.

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

Outline

Basics of lists

`list` is a *sequential* data type in Python – it holds a sequence of "items."

Each item could be an **`int`**, each could be a **`list`**, or possibly some with one type, some with another.

Basics of lists

`list` is a *sequential* data type in Python – it holds a sequence of "items."

Each item could be an **`int`**, each could be a **`list`**, or possibly some with one type, some with another.

Example below: a list with **`int`** and **`str`** type items.

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

Basics of lists

`list` is a *sequential* data type in Python – it holds a sequence of "items."

Each item could be an **`int`**, each could be a **`list`**, or possibly some with one type, some with another.

Example below: a list with **`int`** and **`str`** type items.

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

Referring to a list item by index:

`my_list[0]` is 2 above, `my_list[1]` is 3, and so on.

Basics of lists

`list` is a *sequential* data type in Python – it holds a sequence of "items."

Each item could be an **`int`**, each could be a **`list`**, or possibly some with one type, some with another.

Example below: a list with **`int`** and **`str`** type items.

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

Referring to a list item by index:

`my_list[0]` is 2 above, `my_list[1]` is 3, and so on.

The + operation is defined on lists. 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]
```

Other operations on lists

- Multiplication by an integer: adds that many copies of the list together. For example, `[1,2]*3` will result in `[1,2,1,2,1,2]`, since

$$[1,2] + [1,2] + [1,2] = [1,2,1,2,1,2].$$

Other operations on lists

- Multiplication by an integer: adds that many copies of the list together. For example, `[1,2]*3` will result in `[1,2,1,2,1,2]`, since

  `[1,2] + [1,2] + [1,2] = [1,2,1,2,1,2]`.

- Length of a list: use the function `len()`, with your list as input, to get the number of items in your list.

Other operations on lists

- Multiplication by an integer: adds that many copies of the list together. For example, `[1,2]*3` will result in `[1,2,1,2,1,2]`, since

  `[1,2] + [1,2] + [1,2] = [1,2,1,2,1,2].`

- Length of a list: use the function `len()`, with your list as input, to get the number of items in your list.
- Checking if an item is in a list: use the *keyword* `in` to check this. For example, if `my_list` is `[2, 3, 5, 'p']` then the first line below would result in `True`, the second would be `False`.

```
1 print( 2 in my_list )
2 print( 4 in my_list )
```

Strings and other sequential types

- Some other sequential types: **`tuple`**, **`range`**.
- The operations we discussed (on lists) will work in same way on these.

Strings and other sequential types

- Some other sequential types: **`tuple`**, **`range`**.
- The operations we discussed (on lists) will work in same way on these.
- Final important sequential type: **`str`**, "strings."
  - a sequence of *characters*, from your keyboard
- Above, the variable that was assigned `'Chris Cornwell'`, and the item `'p'` in `my_list`, each is a string.

Strings and other sequential types

- Some other sequential types: **`tuple`**, **`range`**.
- The operations we discussed (on lists) will work in same way on these.
- Final important sequential type: **`str`**, "strings."
  - a sequence of *characters*, from your keyboard
- Above, the variable that was assigned `'Chris Cornwell'`, and the item `'p'` in `my_list`, each is a string.
- *Thinking* of a string as a list of single characters, operations on strings work like they do on lists (e.g., + will concatenate and `len()` gives the number of characters, etc.

f-strings

To produce an output string that uses values of some variables in memory: two methods (*there are others I won't mention*).

`f`-strings

To produce an output string that uses values of some variables in memory: two methods (*there are others I won't mention*).
Say a variable `i` is in memory, with `i = 2`.

f-strings

To produce an output string that uses values of some variables in memory: two methods (*there are others I won't mention*).
Say a variable `i` is in memory, with `i = 2`.

```
1 # the next line will print out 'The value of i is 2 .'
2 print('The value of i is', i, '.')
```

- Works, but has some downsides.

f-strings

To produce an output string that uses values of some variables in memory: two methods (*there are others I won't mention*).
Say a variable `i` is in memory, with `i = 2`.

```
# the next line will print out 'The value of i is 2 .'
print('The value of i is', i, '.')
```

- Works, but has some downsides.
- Better: use so-called `f`-strings, allowing variable(s) directly inside the string, surrounded by braces `{ }`.

```
# the next line will print out 'The value of i is 2.'
print(f'The value of i is {i}.')
```

f-strings

To produce an output string that uses values of some variables in memory: two methods (*there are others I won't mention*).
Say a variable `i` is in memory, with `i = 2`.

```
# the next line will print out 'The value of i is 2 .'
print('The value of i is', i, '.')
```

- Works, but has some downsides.
- Better: use so-called `f`-strings, allowing variable(s) directly inside the string, surrounded by braces `{ }`.

```
# the next line will print out 'The value of i is 2.'
print(f'The value of i is {i}.')
```

Escape characters can be handled inside strings also: e.g., `'\t'` will produce a tab; `'\n'` produces a newline.

Two more container types

Two important types that contain items, but are not sequential are sets (**`set`** type) and dictionaries (**`dict`** type).

- **`set`**: roughly matches the mathematical notion of a set. Items are not ordered; there are no repeated items.

Two more container types

Two important types that contain items, but are not sequential are sets (**`set`** type) and dictionaries (**`dict`** type).

- **`set`**: roughly matches the mathematical notion of a set. Items are not ordered; there are no repeated items.
- **`dict`**: has *dictionary keys*; for each key there is an item (the “entry” for that key).

Two more container types

Two important types that contain items, but are not sequential are sets (**`set`** type) and dictionaries (**`dict`** type).

- **`set`**: roughly matches the mathematical notion of a set. Items are not ordered; there are no repeated items.
- **`dict`**: has *dictionary keys*; for each key there is an item (the "entry" for that key).
- Two example dictionaries with same keys:

```
my_pet = {'name':'Spot', 'age':4, 'type':'dog'}
neighbor_pet = {'name':'Checkers', 'age':2, 'type':'dog'}
```

Two more container types

Two important types that contain items, but are not sequential are sets (**`set`** type) and dictionaries (**`dict`** type).

- **`set`**: roughly matches the mathematical notion of a set. Items are not ordered; there are no repeated items.
- **`dict`**: has *dictionary keys*; for each key there is an item (the "entry" for that key).
- Two example dictionaries with same keys:

```
my_pet = {'name':'Spot', 'age':4, 'type':'dog'}
neighbor_pet = {'name':'Checkers', 'age':2, 'type':'dog'}
```

Good idea to work with dictionaries for certain kinds of data. Later in the semester, will work with something very similar to a dictionary – a DataFrame.

Outline

Basic functions

Similar to most programming languages, Python uses *functions*, each of which takes some number of *arguments* (sometimes an argument is optional).

Basic functions

Similar to most programming languages, Python uses *functions*, each of which takes some number of *arguments* (sometimes an argument is optional).

- Common function, already encountered: `print()`.

Basic functions

Similar to most programming languages, Python uses *functions*, each of which takes some number of *arguments* (sometimes an argument is optional).

- Common function, already encountered: `print()`.
- Absolute value: `abs()`. Takes a numeric argument – either **`int`** or **`float`** type.

Basic functions

Similar to most programming languages, Python uses *functions*, each of which takes some number of *arguments* (sometimes an argument is optional).

- Common function, already encountered: `print()`.
- Absolute value: `abs()`. Takes a numeric argument – either **`int`** or **`float`** type.
- Round to nearest integer: `round()`. Takes argument that is **`float`** (if an **`int`**, will return the same thing).
  - Optional 2nd argument, the number of decimal digits.

Basic functions

Similar to most programming languages, Python uses *functions*, each of which takes some number of *arguments* (sometimes an argument is optional).

- Common function, already encountered: `print()`.
- Absolute value: `abs()`. Takes a numeric argument – either **`int`** or **`float`** type.
- Round to nearest integer: `round()`. Takes argument that is **`float`** (if an **`int`**, will return the same thing).
  - Optional 2nd argument, the number of decimal digits.

In Python, run the following to see how `round()` works.

```
1 a = -3**2/8
2 print( a+8 )
3 print( (round(a+8), round(a+8, 2)) )
```

Changing one type to another

Some functions change the type of the input, when possible.

Changing one type to another

Some functions change the type of the input, when possible.

Some examples:

- `str()` changes to a string; works on nearly anything.

Changing one type to another

Some functions change the type of the input, when possible.

Some examples:

- `str()` changes to a string; works on nearly anything.
- `int()` will convert a float to an int; always rounds toward 0.

Changing one type to another

Some functions change the type of the input, when possible.

Some examples:

- `str()` changes to a string; works on nearly anything.
- `int()` will convert a float to an int; always rounds toward 0.
- `set()` will convert a list or tuple into a set; “forgets” order, drops repeats.

Slicing lists

Recall: if `my_list` is a list, the item at index `i` is found with `my_list[i]`.
To get shorter list with consecutive items from `my_list`.

[2] Only as an output. This does not change the list variable unless you reassign it.

Slicing lists

Recall: if `my_list` is a list, the item at index `i` is found with `my_list[i]`.

To get shorter list with consecutive items from `my_list`.

```
1  my_list = ['a','b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
2  # Return letters at index 1 to 4 (excluding 4)
3  print(my_list[1:4])
4
5  # Leave off number either left of colon, or right of it;
6  # will go from the start, or until the end
7  print(my_list[:5])
8
9  # Negative numbers to step back from end of the list
10 print(my_list[-1])
11 print(my_list[-2:])
```

[2] Only as an output. This does not change the list variable unless you reassign it.

Slicing lists

Recall: if `my_list` is a list, the item at index `i` is found with `my_list[i]`.

To get shorter list with consecutive items from `my_list`.

```
my_list = ['a','b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
# Return letters at index 1 to 4 (excluding 4)
print(my_list[1:4])

# Leave off number either left of colon, or right of it;
# will go from the start, or until the end
print(my_list[:5])

# Negative numbers to step back from end of the list
print(my_list[-1])
print(my_list[-2:])
```

There is an easy way to reverse the order of a list.[2]

```
my_list[::-1]
```

[2] Only as an output. This does not change the list variable unless you reassign it.

Basic list methods

Methods are functions that you call on an instance of a class. There are several methods for lists. Here are two.

Basic list methods

Methods are functions that you call on an instance of a class. There are several methods for lists. Here are two.

- `append()`: the command `my_list.append(x)` puts `x` at the end of `my_list`.
  - Changes `my_list` "in place."

Basic list methods

Methods are functions that you call on an instance of a class. There are several methods for lists. Here are two.

- `append()`: the command `my_list.append(x)` puts `x` at the end of `my_list`.
  - Changes `my_list` "in place."
  - Is equivalent to `my_list += [x]`.

Basic list methods

Methods are functions that you call on an instance of a class. There are several methods for lists. Here are two.

- `append()`: the command `my_list.append(x)` puts `x` at the end of `my_list`.
  - Changes `my_list` "in place."
  - Is equivalent to `my_list += [x]`.
- `remove()`: the command `my_list.remove(x)` takes out the *first* item in `my_list` that is equal to `x`.
  - Changes `my_list` in place, making it shorter.

Basic list methods

Methods are functions that you call on an instance of a class. There are several methods for lists. Here are two.

- `append()`: the command `my_list.append(x)` puts `x` at the end of `my_list`.
  - Changes `my_list` "in place."
  - Is equivalent to `my_list += [x]`.
- `remove()`: the command `my_list.remove(x)` takes out the *first* item in `my_list` that is equal to `x`.
  - Changes `my_list` in place, making it shorter.
  - `my_list.pop(i)` does something similar with item at index `i`, but also returns (has as output) that item.

More information on working with lists, tuples, sets, and dictionaries: Tutorial from the Python documentation.