Tips for CS 1110 students

Ben Rosenberg (bar94)

8/29/2021

Contents

Python tips and conventions . . . 1
- `== True` . . . 1
- Oversimplification . . . 2
- Operator spacing . . . 2
- Long strings . . . 2
- When (and when not) to use `range()` . . . 3
- Format strings (f-strings) . . . 4

Tips for using Atom Editor (and other code editors), and the terminal . . . 4
- Spaces (and tabs) . . . 4
- Comments . . . 5
- Autocomplete and Shell History . . . 5
- Useful commands . . . 5
- The Python REPL (read-evaluate-print loop) . . . 5
- Parting Tips . . . 6

The following is a compendium of small tips for students taking CS 1110: Intro to Computing Using Python at Cornell. In the past year I've picked up on some very common mistakes that students who are new to computing and Python tend to make, and tricks that can help students speed up their workflows considerably. To help students earlier rather than later, I've compiled them here.

Python tips and conventions

`== True`

This is an extremely common thing to find in beginner Python code. The general structure tends to be something like this:

```
# x is a boolean variable (either True or False)
if x == True:
    ... # do something
```

It can be hidden in a larger structure, like this:

```
contains_zero = False
for i in list_of_integers:
    if i == 0:
        contains_zero = True

if contains_zero == True:
    print("There's a zero!")
else:
    print("No zeros here.")
```

Both of these contain the evil `== True` syntax which is a telltale sign of poorly written code. We can rewrite the second of these to not use `== True` as follows:

```
contains_zero = False
for i in list_of_integers:
    if i == 0:
        contains_zero = True

if contains_zero: # <-- no more `== True`!
    print("There's a zero!")
else:
    print("No zeros here.")
```

See the difference? So much cleaner. Please never use `== True`: you can always just delete it and count on Python to use the boolean variable's value as it was intended.

Oversimplification

Speaking of cleaner, we can rewrite this (largely useless) code even more:

```
if 0 in list_of_integers: # Python has a convenient `in` function
    print("There's a zero!")
else:
    print("No zeros here.")
```

This is good – we cut down on the code by 4 lines by using the built-in function `in`.

We could cut down on the size more, but that's not necessarily a good thing:

```
print("There's a zero!" * (0 in list_of_integers) + "No zeros here." * (1 - (0 in list_of_integers)))
```

The first issue here is that it's over 80 characters on a single line, which goes against convention. The second (more important) issue is that it's hard to read, requiring the user to understand the idea of boolean-integer casting, and Pythonic string multiplication. Code which does something on such a simple level as this should be readable by someone whose coding knowledge is only at the level of conditionals.

Not all assignments are code golf!

Operator spacing

A good convention to pick up when writing Python is operator spacing. In this course by convention we put a space on either side of operators:

```
# Good!
a = 1 + 2 + 3 + n

# Bad.
a=1+2+3+n
```

Clearly, the first is more readable.

> *Aside*: In CS 2110, the convention is that "assignment is an asymmetric operator", meaning that we would write `a= 1 + 2 + 3 + n` instead of what we have here. In 1110 we still prefer symmetric spaces around each operator.

Long strings

Another course convention is (as previously mentioned) that we try to limit the length of lines of code to 80 characters. This can make writing long strings difficult, as they can sometimes be over 80 characters by themselves.

To deal with this, there are several ways of splitting up strings to go over multiple lines. As an example, consider the following code that has a line over 80 characters:

```
if letter == 'a':
    out = 'This letter is an "a". Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean sed nu
```

We demonstrate three ways to deal with long strings. Remember to include the spaces when splitting up strings!

Backslashes

```
if letter == 'a':
    out = 'This letter is an "a". Lorem ipsum dolor sit amet, consectetur' \
          ' adipiscing elit. Aenean sed nunc ante. Integer tincidunt, urna '\
          'eget tristique mattis, ipsum velit vulputate lorem, sit amet porta'\
          ' ex sem vitae nunc. Nullam quis metus id ante maximus euismod.'\
          ' Phasellus vehicula risus sed posuere suscipit.'
```

Parentheses

```
if letter == 'a':
    out = ('This letter is an "a". Lorem ipsum dolor sit amet, consectetur'
           ' adipiscing elit. Aenean sed nunc ante. Integer tincidunt, urna '
           'eget tristique mattis, ipsum velit vulputate lorem, sit amet porta'
           ' ex sem vitae nunc. Nullam quis metus id ante maximus euismod.'
           ' Phasellus vehicula risus sed posuere suscipit.')
```

Multi-line (triple-quote) strings *Warning*: This method will include the spacing used over the lines; that is, line breaks will appear in the same way as they are entered in the string. Multi-line strings are best used as code comments.

```
if letter == 'a':
    out = '''This letter is an "a". Lorem ipsum dolor sit amet,
             consectetur adipiscing elit. Aenean sed nunc ante.
             Integer tincidunt, urna eget tristique mattis, ipsum
             velit vulputate lorem, sit amet porta ex sem vitae
             nunc. Nullam quis metus id ante maximus euismod.
             Phasellus vehicula risus sed posuere suscipit.'''
```

As a final tip: notice that the strings are lined up with each other when they go onto multiple lines. This makes reading the code easier on the eyes.

When (and when not) to use `range()`

There are two kinds of for-loops that we talk about in 1110: for-loops that use `range()`, and for-loops that don't.

Here's an example of a `range`-based for-loop:

```
for i in range(6):
    print(i)
```

Here's the same loop without using `range()`:

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

Obviously, in this case, it's better to use `range()`. We want direct access to the numbers, and enumerating them by hand in a list is a pain.

But consider this example:

```
letters = ['a', 'b', 'c', 'd', 'e']

# using range()
for i in range(len(letters)):
    print(letters[i])

# not using range()
for i in letters:
    print(i)
```

Clearly, `range()` isn't as good here. We don't need to access the numbers, and referring to the elements of `letters` using their indices is just extra work.

Of course, you could have cases where you want to access the indices while looping through a list too:

```
letters = ['a', 'b', 'c', 'd', 'e']

# using range()
for i in range(len(letters)):
    print(letters[i] + ' is the ' + str(i) + 'th letter in the list')

# not using range()
index = 0
for i in letters:
    print(i + ' is the ' + str(index) + 'th letter in the list')
    index = index + 1
```

In order to not use `range()` in the above example, we need to use the variable `index` as an accumulator to keep track of the current index, which is unwieldy. So, we prefer to use `range()`.

> *Aside*: Since we can't use `range()` in a while-loop, accumulators are the primary way that we iterate and keep track of indices.

Format strings (f-strings)

In the above example, our method of printing the desired string was rather complicated:

```
...
print(letters[i] + ' is the ' + str(i) + 'th letter in the list')
...
```

We can remedy this with the use of f-strings:

```
...
print(f'{letters[i]} is the {i}th letter in the list')
...
```

Simpler and cleaner, and doesn't require any casting to string.

Tips for using Atom Editor (and other code editors), and the terminal

Spaces (and tabs)

In order to use Atom (or whatever other code editor you're using) correctly with Python, you need to be indenting with spaces and not tabs. We recommend that you indent with 4 spaces, and not 2 or 8.

You can set Atom to use spaces instead of tabs by going to Settings (either `Ctrl + ,` or `Command + ,`) > Editor > Tab Type and choosing "soft". The setting for tab length is directly above that (4 recommended).

Comments

Sometimes you'll need to comment out a large portion of code, when you're testing to see if it's causing errors or just don't want to run it for other reasons. Most code editors (Atom, VSCode, and Eclipse all have it, and I assume others do too) will have some kind of shortcut to comment out selected lines so that you don't have to painstakingly go through and comment each of them out. Usually this is `Ctrl + /` (or `Command + /` on Mac). This command will save you a lot of time.

Autocomplete and Shell History

When you're in the terminal it can be a pain to type out the entire file name when you want to run it, or the entire folder name when you want to `cd` to it. Most terminals have a feature where you can type out the first character of a name, and then hit `Tab` to autocomplete. If there are multiple items in the current directory with the same first character, usually you can cycle through them by hitting `Tab` again.

Another common thing that you'll want to do in the terminal is rerun your most recently run command (or second or third or fourth most recently run command). To scroll back through your terminal history, hit the up arrow on your keyboard. As with `Tab`, repeatedly hitting the up arrow will scroll through older and older commands that you have run.

Useful commands

Here's a (short) list of useful commands:

command	purpose	usage
`cd`	change directory	`cd directory`
`pwd`	print working directory	`pwd`
`ls`	print contents of working directory	`ls`, `ls -l`, `ls -a`
`mv`	move files/folders	`mv file.ext new_location`
`rm`	delete file/folder	`rm file.ext`, `rm -rf folder`

The Python REPL (read-evaluate-print loop)

The Python REPL or interactive shell looks like this:

```
>>>
```

You can type things in here to be instantly evaluated, making it a convenient way to test things in Python (or use Python as a calculator).

Exiting the REPL Some people have trouble exiting the Python REPL. Here are some ways:

- `Ctrl + D`: may or may not work. Usually works on UNIX-based OSes (Mac and Linux); I don't think it works in Powershell.
- `exit()`, `quit()`: type one of these and hit Enter. Works 100% of the time.

If you want to stay in the REPL but want to quit a currently running command (e.g., if you accidentally made an infinite loop) use `Ctrl + C` to kill the currently running process while staying in the environment.

Running code from within the REPL It's not recommended to run modules that you write from within the REPL. Instead, do `python my_module.py` to run your code. That way, you can rerun your code easily (with the up arrow as previously mentioned).

If you do want to run your code in the REPL for debugging purposes or some other reason, do

```
>>> import my_module
```

Remember *not* to include the `.py` extension.

Aside: (Not taught in CS 1110) To make certain parts of your code *not* run when you import the module, you can make a section as follows:

```
if __name__ == "__main__":
    ...
```

This section will only run when the code is *not* imported (it will only run when the file is run directly). This can allow you to import things from a module and also run it for different levels of functionality.

Parting Tips

Lastly: remember to **run your code**! There is a "crash tax" of 5 points (which, when combined with the errors that made your code crash, leads to a larger deduction) regardless of the assignment's total number of points, given to students whose code fails to execute without errors. On some assignments, the point total may be as low as 50 or 60, meaning that you could lose around 10% of your total score this way!

Good luck, and happy coding!