Notebook

March 30, 2019

1 Dealing with Errors

An unavoidable fact of life for people who write code is error messages. You're happily programming along, and then, all of a sudden, you get a massive screen of terrifying text telling you that *you screwed up!!*

```
In [1]: print(["cat", "dog", "bear"][3])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)

<ipython-input-1-642bdd2024de> in <module>
----> 1 print(["cat", "dog", "bear"][3])


IndexError: list index out of range
```

The first thing you should do when you see an error is *be grateful*. Error messages are a good thing. Because you've asked the computer how to do something that it doesn't understand how to do, and if you don't get an error message, either the program would have to

a) silently fail—that is, not do anything, and not tell you why, making it much harder to debug when your program doesn't work, or

b) much, much worse: make some kind of guess about what you're trying to do, and do that.

Option b in particular can be a massive disaster. For example, the error generated by the code above was a result of my asking the list to fetch an item that it didn't exist. Remember: Python indices start at 0, so in `["cat", "dog", "bear"]` the only indices that exist are 0, 1, and 2.

Some languages will actually let you access the index of their list-like data structures that is out of their range, and then just give you the contents of whatever is in memory at that location—which could (if the operating system sucks) be memory belonging to a different program, or it could be information that you don't want to be accessed in the current program.
Imagine, for example, that your program just holds a list of user information and also holds passwords, and you have some function that users can call to get at the user information, but you

accidentally started counting at 1 instead of 0 (a common "off-by-one error") when you wrote the program. All of a sudden, users try to get the last item in their list, and they get a password back instead... maybe someone else's! This is a contrived example, but it's not all that different from stuff that really happens in languages like C.

So, errors are awesome. (There are still times your programs will go wrong without helping you out with errors, and we'll talk about figuring out what's going on there toward the end of this lesson, but let's start at the start.) Let's figure out how to deal with them.

First of all, **look at the type of the error, which is at the very bottom of the error message.** Typically the last line is always the first place to look, and, in the error above, it should be pretty obvious what happened once you see it. IndexError means, as expected, that you tried to use an index that doesn't exist. Note that up above we helpfully get the line that caused the error.

Let's look at some other common errors.

1.1 Easy errors

```
In [2]: for x in ["cat", "dog" "bear"]:
            print(x)

cat
dogbear
```

Hah! That was supposed to throw an error, but it turns out that Python has bizarre behavior where if you leave off a comma in a list of strings it just concatenates them. I literally discovered that while writing this lesson. I'll leave it here as an object lesson in why errors are good: if this happened to you in a real program you'd probably get subtly wrong behavior that would be hard to figure out.

```
In [3]: for x in [1, 2 3]:
            print(x + 1)


          File "<ipython-input-3-08a169b2bac7>", line 1
        for x in [1, 2 3]:
                       ^
    SyntaxError: invalid syntax
```

Ok, that's more like it! So what we see here is a syntax error. That's a nice easy error to fix, because once you learn the language, it's usually just because of a typo.

Note how the second-to-last line has a convenient little caret (arrow thing) pointing to the spot in the line *after* where the typo was? That's sort of a normal thing for this kind of error: Python can't actually figure out where the typo is, but it can figure out where it all of a sudden started getting data in a format it didn't know how to process, so it shows you that.

```
In [4]: print("the cat " + "in " + "the hat)
```

```
  File "<ipython-input-4-f2a1d13896e4>", line 1
    print("the cat " + "in " + "the hat)
                                        ^
SyntaxError: EOL while scanning string literal
```

Another common typo: you forgot to put the closing quotation mark in (or you put the wrong quote in, as in the following example:

```
In [5]: print("the cat " + "in " + 'the hat")
```

```
  File "<ipython-input-5-0277e4720369>", line 1
    print("the cat " + "in " + 'the hat")
                                       ^
SyntaxError: EOL while scanning string literal
```

Again, if you think through the logic of how Python's computer brain works, you can make sense of the error. Python saw the opening of a string (in this case, `'the hat`) but didn't see the end of it. And it got to the end of the line, but it was still looking for more string characters, because it never got a close quote indicating that the string was over! Hence "EOL while scanning string literal"

The different colors in the code in the Jupyter notebooks, incidentally, will often help you figure out this kind of syntax error. Do you see how strings tend to be colored red? When you see that `'the hat")` is all red in your syntax highlighter, that should be a clue: the close parens is normally black!

```
In [6]: for x in range(5):
            for y in range(3):
            print(x * y)
```

```
  File "<ipython-input-6-845bd4ebbfb0>", line 3
    print(x * y)
        ^
IndentationError: expected an indented block
```

There's another easy one: you forgot to indent when you were supposed to. Pretty straightforward.

```
In [7]: def my_func(data):
            output = []
            for x in data:
```

```
            for y in range(3):
                z = x * y
                output.append(z)
        return output

    # verify the function works as expected before we start blowing it up:

    my_func(range(5))
```

```
Out[7]: [0, 0, 0, 0, 1, 2, 0, 2, 4, 0, 3, 6, 0, 4, 8]
```

```
In [8]: my_fuc(range(5))
```

```
    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-8-344d0251a0a9> in <module>
    ----> 1 my_fuc(range(5))


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

NameErrors are often going to be typoes. Sometimes, however, they'll be scope errors, as in the following:

```
In [9]: result = my_func(range(10))
        print(output)
```

```
    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-9-cc741f8cd5ed> in <module>
      1 result = my_func(range(10))
    ----> 2 print(output)


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

This looks like forgetting that `output` only exists within the scope of the function.

1.2 Slightly more difficult errors

Here's one that I screw up *all the time*. (You'll probably see me do it at least once in class this semester.

```
In [10]: unsorted_list = [3, 4, 1, 6, 9, 10, 5, 2, 8, 7]

         for x in unsorted_list.sort():
             print(x)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-10-505a71766970> in <module>
          1 unsorted_list = [3, 4, 1, 6, 9, 10, 5, 2, 8, 7]
          2
    ----> 3 for x in unsorted_list.sort():
          4     print(x)


        TypeError: 'NoneType' object is not iterable
```

What happened here? Take a couple minutes and think about it. Then try some code of your own to see if you can make the error go away.

.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.

hint: go back and look over the complex data types lesson from last week. What does the sort method on lists do?

.

.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.

Answer coming soon. Have you thought about it some? Have you tried out some code to see if you can figure out how to get it to work?

.

.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.

And the answer is: the `sort()` method of lists doesn't return a new, sorted list, it sorts the original list in place. So the code `for x in unsorted_list.sort():` evaluates `unsorted_list.sort()` to `None`, which is what methods that just mutate in place return. This means that the code I ran really is equivalent to:

```
for x in None:
    print(x)
```

which obviously makes no sense.

Worse, because you've mutated `unsorted_list` it is no longer, well, unsorted, and if you were expecting to have access to its original, unsorted, form later, you will be unhappily surprised.

```
In [11]: print(unsorted_list)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

incidentally, here's the right way to do that: use the `sorted()` function, which returns a new list and doesn't mutate the original

```
In [12]: unsorted_list = [3, 4, 1, 6, 9, 10, 5, 2, 8, 7]

         for x in sorted(unsorted_list):
             print(x)
         print(unsorted_list)
```

```
1
2
3
4
5
6
7
8
9
10
[3, 4, 1, 6, 9, 10, 5, 2, 8, 7]
```

Generally, whenever you see TypeErrors involving `NoneType` it means that something you thought was going to return a value actually didn't. Very often, that's because of functions that don't return anything. Sometimes it's also because of functions that were supposed to return something, but couldn't find anything to return, like network requests.

There are other kinds of type errors though.

```
In [13]: my_func(3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-13-5c6f14bd9907> in <module>
----> 1 my_func(3)


<ipython-input-7-0985c578d18e> in my_func(data)
  1 def my_func(data):
  2     output = []
----> 3     for x in data:
  4         for y in range(3):
  5             z = x * y


TypeError: 'int' object is not iterable
```

Again, look at the last line first. Remember that `my_func` loops over something to try to multiply everything in it by everything in `[0, 1, 2]`. So you gotta give it something you can loop over. And you can't loop over the number 3, this ain't Sesame Street.

I often get this kind of error by accidentally putting in the name of a function that doesn't take any arguments rather than calling it, i.e., typing `some_fancy_function + 5` when I mean to type `some_fancy_function() + 5`

```
In [14]: my_func("cat", "dog")
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-8a28da907df6> in <module>
----> 1 my_func("cat", "dog")


TypeError: my_func() takes 1 positional argument but 2 were given
```

This is also a TypeError (don't ask me why): wrong number of arguments to a function.

```
In [15]: my_func()
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-15-db3ada79940f> in <module>
----> 1 my_func()


TypeError: my_func() missing 1 required positional argument: 'data'
```

Ditto.

```
In [16]: my_func(["cat", "dog"])

Out[16]: ['', 'cat', 'catcat', '', 'dog', 'dogdog']
```

That is *not* an error, even though it probably wasn't what you were expecting. The reason is that the * symbol is *overloaded* in Python: you can use it to multiply numbers, or, if you do string * a number, you get repeated strings. This is an important reminder: not every mistake will throw an error!

```
In [17]: "kitty" * 10

Out[17]: 'kittykittykittykittykittykittykittykittykittykitty'

In [18]: my_func([True, False])

Out[18]: [0, 1, 2, 0, 0, 0]
```

Another situation where you would expect an error, but you don't get one. Basically, what's happening here is that Python is silently coercing `True` to 1 and `False` to 0 so that they have compatible types to the thing you're trying to * them with, namely, integers.

This is, arguably, pretty bad behavior. Other languages are much stricter about the types of variables and don't let you make this kind of mistake. But the tradeoff is that those other languages are harder to program in—they make you follow lots more rules in order to get working code.

1.3 Actually difficult errors

I think there are two kinds of really common errors that can take forever to deal with: accidentally renaming things and errors generated from deep within complicated libraries. (There are also many, many uncommon errors that you will see.

Both of those are actually fairly difficult to show here, so we'll approach them when we see them in class (or the first time someone generates one of them out of the classroom)... watch this space for more.

In the meantime, let's talk strategies for fixing silent errors. Often times, you'll find out that your code is subtly wrong, and you won't know why—it'll just be producing the wrong answer. Sometimes a freakishly wrong answer.

The way to think about this task is as an attempt to narrow down where you made the mistake. For example, suppose you have some complicated buggy function. Let's say it does some math, and you're mysteriously getting the wrong answer.

```
In [19]: def complicated_sum_all_in_list(nums):
             length_of_list = len(nums)
             first_half_of_list = nums[1:round(length_of_list/2)]
             second_half_of_list = nums[round(length_of_list/2):]
             answer = 0
             for x in first_half_of_list:
                 answer += x  # this is just shorthand for answer = answer + x. it's not the bug
             for x in second_half_of_list:
                 answer += x
             return answer

In [20]: complicated_sum_all_in_list([1, 2, 3, 4, 5])

Out[20]: 14
```

14? That doesn't look right. And we can verify that it isn't right by using a non-idiotic way of summing the list, which happens to be built right into Python:

```
In [21]: sum([1, 2, 3, 4, 5])

Out[21]: 15
```

Ok, so how did we get the math wrong? Other than by writing an idiotically complicated function. Let's assume we can't just look at our code and see what we did. (Experienced coders will be able to look at that one and see, it's a pretty basic oops.) What we'd ideally like to to is be able to look inside the execution of the function and see what's happening.

Fancy programmer types use what's called a debugger to do this—basically a debugger will let you pause your code at any point, and see what the values of the variables are and so forth. See this blog post for more on how to use them in Jupyter Notebooks if you want.

but really, most of the time, simple calls to `print()` will help. Let's dig into our function with them.

```
In [22]: def complicated_sum_all_in_list(nums):
             length_of_list = len(nums)
```

```
            first_half_of_list = nums[1:round(length_of_list/2)]
            second_half_of_list = nums[round(length_of_list/2):]
            answer = 0
            for x in first_half_of_list:
                answer += x  # this is just shorthand for answer = answer + x. it's not the bug
                print(answer)
            for x in second_half_of_list:
                answer += x
                print(answer)
            return answer
        complicated_sum_all_in_list([1, 2, 3, 4, 5])
```

```
2
5
9
14
```

Out[22]: 14

We can get a bit more of a clue what's going on already. We're using print calls to see the incremental answers (sums) being generated by our program, and it looks like the first one is wrong. Why is the incremental sum starting at 2? Shouldn't 0 + 1 be equal to 1?

Let's try more print statements.

```
In [23]: def complicated_sum_all_in_list(nums):
            length_of_list = len(nums)
            first_half_of_list = nums[1:round(length_of_list/2)]
            second_half_of_list = nums[round(length_of_list/2):]
            answer = 0
            for x in first_half_of_list:
                print(x)
                answer += x  # this is just shorthand for answer = answer + x. it's not the bug
            for x in second_half_of_list:
                print(x)
                answer += x
            return answer
        complicated_sum_all_in_list([1, 2, 3, 4, 5])
```

```
2
3
4
5
```

Out[23]: 14

Now what we tried to do is print the numbers that are being added up, and we can see that it's starting with the wrong number. It's taking the list `[1, 2, 3, 4, 5]` and only adding `[2, 3, 4, 5]` together.

This should give you enough of a clue to figure out what went wrong in the code. So here's a mini-homework assignment: fix the error in that function!