Kartik's *Exceptional* Guide to Python Error Messages

Kartik Chandra

CS 106AP, Spring '19

Introduction

Python's error messages can be pretty cryptic. This is a handy pocket guide to dealing with them.

The recommended usage is (1) read the section below fully to get some background information on error messages, and then (2) search for your specific error in this PDF (using either the table of contents or cmd-F) and read the relevant section to figure out what's going on in your case. Good luck, and happy debugging!

Anatomy of an error message

Here is a (partially colorized) example of an error message after running some Python program `getdata.py` with some input `data.txt`. This is admittedly not pleasant to read, but there is a lot of valuable information here if you know where to find it.

```
$ python get_data.py data.txt
Traceback (most recent call last):
  File "getdata.py", line 13, in <module>
    all_numbers.extend(getnumbers(line))
  File "getdata.py" , line 6 , in getnumbers
    numbers.append(int(num))
ValueError : invalid literal for int() with base 10: 'a'
```

The trick is to read the error message *bottom to top* — this is what "most recent call last" means. The first thing you should read is `ValueError`, which tells you what name of the error you are dealing with. After that there is the message "invalid literal for int() with base 'a'" which provides a clue about what is wrong. This message is supposed to be human-friendly but typically fails to achieve that goal.

Above that is the offending line of code, which is literally the actual line in your program that triggered the error! If you are having trouble finding it, the information preceding it tells you the file where the line is ("getdata.py"), the line number within the file (6), and the function in which the offending line occurs (`getnumbers`).

If you wanted *even more* information, you can read the next line up, which tells you how you got to the current function (here, the answer is that `getnumbers(line)` was called on line 13 of "getdata.py").

Contents

1 AssertionError

This error means that an assert statement failed. If *you* put in the assertion, you should already know what went wrong! If the assertion is in someone else's code, then check the error message for more information. This should very rarely happen in CS 106AP.

```
>>> print(x)
1
>>> assert x == 2, "x␣should␣equal␣2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: x should equal 2
```

Debugging strategies: Check the error message.

2 AttributeError

This error means you tried to use the 'dot' syntax to access something that does not exist.

```
>>> a = [1,2,3]
>>> a.appedn(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'appedn'
```

In this example, lists don't have any functionality called 'appedn' (it's a typo for 'append'!) so Python gets confused.

Debugging strategies: Is there a typo in the word after the dot? Is the variable before the dot exactly what you think it is? Print out the variable before the dot ('a' in the example above) to double-check.

3 FileNotFound

This error means that you tried to read a file that does not exist.

```
>>> with open('no-file.txt') as f:
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'no-file.txt'
```

Debugging strategies: Check the error message for the filename. Does it look right? If so, check that the file is in the correct directory — it should be in the same directory as where the Python program is.

4 ModuleNotFoundError

This error means exactly what it says: you tried to import a module that was not found.

```
>>> import banana
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'banana'
```

Debugging strategies: Is there a typo in the module name? Is the module installed correctly? Debugging module installations can be tricky; this is a good place to ask an SL for help.

5 IndexError

This error means that you tried to index (using square brackets `[]`) with an index that is too big.

```
>>> a = [1, 2, 3]
>>> a[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Debugging strategies: Print out the index! A common source of this bug is an off-by-one error, where the last iteration of a loop takes you one past the last element in the list.

6 KeyError

This error means you are trying to look for a key that does not exist in a dictionary.

```
>>> a = {'banana': 3}
>>> a['apple']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'apple'
```

Debugging strategies: Look at the error message, which tells you the key that you looked up. Maybe you want to first check if a key is in the dictionary before accessing it, and set it to some default value if the key is not found (this is a very common pattern).

7 KeyboardInterrupt

This oddly-named error[1] means that the program ran for a long time until *somthing* stopped it (either you stopped it forcibly, or PyCharm killed it for some reason, or it timed out).

```
>>> while True:
...     pass
... # now I forcibly killed the program
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Debugging strategies: This typically indicates either an infinite loop or a program that is doing too much work (for example, reading in a large file over and over again instead of reading it once and saving the data). Add some print statements at the beginning of each suspicious-looking loop and see where it gets stuck.

[1]The name comes from the fact that typically programs are killed by using the control-C keyboard combination... but that is by no means the only way to terminate a running a Python program!

8 NameError

This error means that you tried to get the value of a variable that does not exist.

```
>>> unknownvariable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'unknownvariable' is not defined
```

Debugging strategies: Is there a typo in the variable name? Did you intend to define the variable earlier (i.e. using the '=' operator)? Is there a typo where you *defined* the variable (this includes function arguments)? Are you using a variable from another function?

9 RecursionError

This error is weird: it means that you wrote a function that calls *itself* and gets stuck in a loop that way! It's also possible that you have a chain of functions that call each other causing a loop.

A function calling itself is a valid programming technique, but it is an advanced topic that CS 106B covers — in CS 106AP you shouldn't worry too much about this.

```
>>> def stupid():
...     stupid()
...
>>> stupid()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in stupid
  File "<stdin>", line 2, in stupid
  File "<stdin>", line 2, in stupid
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Debugging strategies: The error message gives you the name of the bad function. Why does the function call itself?

10 SyntaxError

This error means your “syntax" is wrong. Typically this is a typo (e.g. missing colon).

```
>>> while True
  File "<stdin>", line 1
    while True
             ^
SyntaxError: invalid syntax
```

Debugging strategies: The little ‘^’ sign indicates the place where Python thinks the problem is (in the example above, it’s pointing to the end of the `while True` line because there is no colon). You should also double-check your indentation (each level should be +4 spaces). It helps to get a second pair of eyes to go over the code sometimes, just like trying to find spelling mistakes in an essay. One last tip is that sometimes the syntax error is actually in the line immediately *before* or immediately *after* the line that Python reports, so it’s worth finding the line directly in your code.[2]

[2]In fact, reporting ‘good’ error messages for syntax errors is a hard research problem!

11 TypeError

This error typically means that you are giving some operation the wrong types of arguments.

```
>>> 1 + 'cow'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> for i in 5:
...     print(i)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Debugging strategies: Python doesn't tell you which value is of the wrong type, so the first step is to identify the type of each value in your code. Maybe you are missing a call to `int` or `str` somewhere. Make sure you're doing for/in only on things that you can loop over such as lists, dicts, and files (this is what 'iterable' means). Errors involving `NoneType` usually mean that you forgot a `return` statement in some *other* function, which means 'nothing' got returned.

12 UnicodeDecodeError

This error usually has something to do with trying to read a file that is *not* plain-text data. For example, it might have special characters (like accénts or s¥mbols, or even ‘smart quotes’ — or long dashes). Or it might actually be a binary file like an image or video, which doesn’t make sense to read in as ‘text.’

```
>>> with open('my-amazing-selfie.png', 'r') as f:
...     for line in f:
...             pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/usr/local/lib/python3.7/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, ...
UnicodeDecodeError: 'utf-8' codec can't␣decode␣byte␣0xff...
```

Debugging strategies: Make sure you’re opening the correct file, and that it doesn’t have accents, etc. If you are actually trying to read a file with special characters, consult an SL for how to fix the encoding issue. Typically using a variant of `open` such as `open(filename, encoding='utf-8')` will work.

If you are actually trying to read a *binary* file (like an image), use `'rb'` instead of just `'r'` as the second argument to `open` (the ‘b’ stands for ‘binary,’ as you may have guessed).

13 ValueError

This error is kind of a wildcard, because many things could have gone wrong. Typically it means you gave a function the right type (so not a `TypeError`) but an invalid value. A good example is calling `int` with a string that does not represent a number.

```
>>> int('cheese')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'cheese'
```

Debugging strategies: Typically this needs to be handled on a case-by-case basis. Common causes are bad calls to `int`, unpacking a tuple with the wrong number of variables (it should always be the same as the size of the tuple), opening a file with a mode that isn't valid (e.g. 'r' and 'w' are valid but 'q' is not), and so on.

14 ZeroDivisionError

This error means you are asking Python to divide by zero.

```
>>> 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Debugging strategies: Print out the number you are dividing by. Trace back to see why it is zero. Do you have an off-by-one error somewhere?