Chapter 9

Dealing With Errors in Python

Skills you will learn: How to identify, correct, and handle syntax errors, exceptions, and logical errors in Python scripts.

Introduction: One of the most frustrating aspects of working with any programming language is dealing with errors, the glitches that stop our scripts from running, or produce illogical or mal-formed output. Part of the frustration stems from the fact that computers are ultimately stupid machines. For sure they are machines that can process vast amounts of data far faster than we could by hand, but to do that they need precise instructions. Those come from the programs that we run on the machines.

Unlike instructions that we may give to fellow humans, instructions given to computers have to be formulated in exactly the right way or the computers will fail at the tasks we give them, reporting errors in the process. Tracking down, identifying and correcting those errors is a process known as "debugging" and it is an expected part of the routine of programming. The more experienced you get the better you will get at it, and tools offered in more advanced coding editors and integrated development environments (see the tutorial **Choosing a Code Editor**) can make the job far easier.

We're going to take a very brief look at errors and basic debugging. But there are plenty of online and written resources if you want to learn more sophisticated techniques.

Syntax Errors

Syntax errors are errors that are caused by mistakes in the way code is written. These are the simplest types of errors. You simply typed something the wrong way, and Python said, "whoops!"

Take the following code:

```
myVariable = "Hello World"

prnt myVariable
```

Try to run it, and Python quits with this error message:

```
File "C:/Users/Owner/PycharmProjects/Teaching/test.py",
line 2
    prnt myVariable
                 ^
SyntaxError: invalid syntax
```

Python is telling us that in line 2 of the script we wrote something in a way it didn't understand. Looking carefully, we see that we wrote prnt myVariable instead of print myVariable

Easy enough to fix.

Exceptions: Exceptions occur when code that is otherwise syntactically correct, generates an error when we try to run it. These are slightly more challenging to debug, because we have to figure out why the error is occurring. Again, though, Python reports the error in a way that can help us.

Let's take the following simple script as an example.

```
1. myCities = ["San Francisco", "Miami", "New York"]

2. myAges = [25, 35, 55]

3. for x in range(0, len(myCities)):

4. mySentence = "I live in " + myCities[x] + " and my age is
   " + myAges[x]

5. print mySentence
```

This is not an uncommon operation; we are concatenating some strings with some variables to produce an output. Running the script, however, produces this error at runtime:

```
Traceback (most recent call last):
  File "C:/Users/Owner/PycharmProjects/Teaching/test.py",
line 4, in <module>
    mySentence = "I live in " + myCities[x] + " and my age
is " + myAges[x]
TypeError: cannot concatenate 'str' and 'int' objects
```

This is Python's way of saying, "I understand what you want to do, but you can't do it." The good thing is that within this rather cryptic output is the answer to what we did wrong.

First, the error message tells us that the error is likely found in line 4 of the script. It helpfully prints out the part of the script that seemed to generate the error. Sometimes the actual error will be on a line before or after, but nonetheless, this is a big help.

The error message goes on to tell us that the problem is a type error; it seems that we were trying to concatenate, or join together, an object of type 'str' and another of type 'int.' That's not something you can do, because to the computer, a string and an integer are two incompatible things. They are different data types, and they can't dance.

Let's look at what the suspect line is doing:

```
mySentence = "I live in " + myCities[x] + " and my age is "
+ myAges[x]
```

It creates a new variable, called mySentence, and assigns to it the outcome of the concatenation operation that joins the string "I live in " to each element in the myCities list, then to the string " and my age is " and then to each element of the list myAges. If we look back at the place in the code where we declared the two lists we can see the issue:

```
1. myCities = ["San Francisco", "Miami", "New York"]

2. myAges = [25, 35, 55]
```

The second list, myAges, is a list of integers. Everything else in the concatenation

operation is a string, but here we are trying to join an integer to all those strings. The result is the type error.

Fortunately, we can fix the problem pretty easily; we'll wrap each element from the myAges list in the built-in Python str() function. It will convert the integer to a string. So we'll rewrite the suspect line like this:

```
mySentence = "I live in " + myCities[x] + " and my age is "
+ str(myAges[x])
```

Now, the script works as expected, with this output:

```
I live in San Francisco and my age is 25
I live in Miami and my age is 35
I live in New York and my age is 55
```

And that is the essence of debugging. Identify the error, find it in the code, and apply the fix.

A list of possible exceptions can be found at: https://docs.python.org/2.7/library/exceptions.html#bltin-exceptions

Logical Errors

As implied by the name, logical errors are produced by errors in logic. The code is syntactically correct, and doesn't produce any errors, but the outcome isn't what you expected. These can be tougher to root out because you have to figure out why the script is behaving the way it is.

Let's say you were using the Beautiful Soup module to grab all of the <tr>, or table row, elements from an html page, using code such as this:

```
soup = BeautifulSoup(html,'html.parser')
theRows = soup.findAll('tr', class_ = "datarow")
for row in theRows:
    theCells = row.findAll('td')
    for cell in theCells:
        print cell
```

Now, say you run the code and you get this output and it completes without reporting any errors, but without printing anything either. What's going on?

This can be the most frustrating kind of error to track down because there is no indication of what's wrong. You need to put on your deerstalker and start sleuthing. The problem could be anywhere in the code. Perhaps the findAll method in the second line of code is not finding any <tr> tags with the class of "datarow" and is thus returning an empty resultset (list) object. Or perhaps there are no <td> tags in the rows so when the second for loop tries to loop through theCells, there's nothing to iterate through.

One simple debugging approach is to use print statements after each line that produces some output.

For example, if one added the statement print theRows after the second line of code, one would be able to see if there was anything in the resultset. If the output from printing it was [] , that is, an empty resultset, or empty list, then one would have the answer. For some reason line 2 was not finding anything. Perhaps the class has a different name, or there are no table rows.

On the other hand, if the print statement produced a resultset with table rows in in it, one could move on to print out theCells, which is similarly the result of calling the Beautiful Soup findAll method. If it turned out to produce an empty list, then one could divine that the problem lay there.

In this way, you can work through your code, and root out logical errors. A logical error may also cause an exception as, for example, the unexpected contents of a variable then trip up a subsequent statement that is supposed to process that variable in some way.

Error handling: Every programming language has some way to handle syntax errors so that rather than causing the script to fail, the error is processed in some way, and the script continues on its way. Python's error handling method is the try/except clause. Let's go back to the code we used when we discussed exceptions.

```
1. myCities = ["San Francisco", "Miami", "New York"]

2. myAges = [25, 35, 55]

3. for x in range(0, len(myCities)):

4. mySentence = "I live in " + myCities[x] + " and my age is
   " + myAges[x]
```

```
5. print mySentence
```

If we expect that part of our code may sometimes generate an error, we can use try/except:

```
try:

     mySentence = "I live in " + myCities[x] + " and my age
is " + myAges[x]
print mySentence

except:

     print "Oops, that didn't work."
```

Now, instead of the script quitting when it encounters the integer where it expected a string, it would print "Oops, that didn't work."

When Python encounters a try/except structure, it first tries to run the code in the try block. If that code runs OK, then nothing more happens. But if it generates an exception, the error is "trapped" and the Except clause runs. In this case, it prints out "Oops, that didn't work."

Try/except is far more sophisticated than this, however. You can also get it to report back what the error was:

```
try:

     mySentence = "I live in " + myCities[x] + " and my age
is " + myAges[x]
print mySentence

except Exception, e:

     print e
```

When the try block fails, the exception, or error, code is stored in the variable e. It is then printed.

It is also possible to take specific action beyond printing the errors, such as running alternative code. If you want to dive into the nitty gritty, there is excellent documentation at https://docs.python.org/2.7/tutorial/errors.html