software carpentry

Programming with Python

Errors and Exceptions

Learning Objectives

- To be able to read a traceback, and determine the following relevant pieces of information:
  - The file, function, and line number on which the error occurred
  - The type of the error
  - The error message
- To be able to describe the types of situations in which the following errors occur:
  - `SyntaxError` and `IndentationError`
  - `NameError`
  - `IndexError`
  - `FileNotFoundError`

Every programmer encounters errors, both those who are just beginning, and those who have been programming for years. Encountering errors and exceptions can be very frustrating at times, and can make coding feel like a hopeless endeavour. However, understanding what the different types of errors are and when you are likely to encounter them can help a lot. Once you know *why* you get certain types of errors, they become much easier to fix.

Errors in Python have a very specific form, called a traceback. Let's examine one:

```
import errors_01
errors_01.favorite_ice_cream()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-9d0462a5b07c> in <module>()
      1 import errors_01
----> 2 errors_01.favorite_ice_cream()

/Users/jhamrick/project/swc/novice/python/errors_01.pyc in favorite_ice_cream()
      5         "strawberry"
      6     ]
----> 7     print(ice_creams[3])

IndexError: list index out of range
```

This particular traceback has two levels. You can determine the number of levels by looking for the number

of arrows on the left hand side. In this case:

1. The first shows code from the cell above, with an arrow pointing to Line 2 (which is `favorite_ice_cream()`).
2. The second shows some code in another function (`favorite_ice_cream`, located in the file `errors_01.py`), with an arrow pointing to Line 7 (which is `print(ice_creams[3])`).

The last level is the actual place where the error occurred. The other level(s) show what function the program executed to get to the next level down. So, in this case, the program first performed a function call to the function `favorite_ice_cream`. Inside this function, the program encountered an error on Line 7, when it tried to run the code `print(ice_creams[3])`.

> 📌 Long Tracebacks
>
> Sometimes, you might see a traceback that is very long – sometimes they might even be 20 levels deep! This can make it seem like something horrible happened, but really it just means that your program called many functions before it ran into the error. Most of the time, you can just pay attention to the bottom-most level, which is the actual place where the error occurred.

So what error did the program actually encounter? In the last line of the traceback, Python helpfully tells us the category or type of error (in this case, it is an `IndexError`) and a more detailed error message (in this case, it says "list index out of range").

If you encounter an error and don't know what it means, it is still important to read the traceback closely. That way, if you fix the error, but encounter a new one, you can tell that the error changed. Additionally, sometimes just knowing *where* the error occurred is enough to fix it, even if you don't entirely understand the message.

If you do encounter an error you don't recognize, try looking at the official documentation on errors. However, note that you may not always be able to find the error there, as it is possible to create custom errors. In that case, hopefully the custom error message is informative enough to help you figure out what went wrong.

Syntax Errors

When you forget a colon at the end of a line, accidentally add one space too many when indenting under an `if` statement, or forget a parenthesis, you will encounter a syntax error. This means that Python couldn't figure out how to read your program. This is similar to forgetting punctuation in English:

> this text is difficult to read there is no punctuation there is also no capitalization why is this hard because you have to figure out where each sentence ends you also have to figure out where each sentence begins to some extent it might be ambiguous if there should be a sentence break or not

People can typically figure out what is meant by text with no punctuation, but people are much smarter than computers. If Python doesn't know how to read the program, it will just give up and inform you with an error. For example:

```
def some_function()
    msg = "hello, world!"
    print(msg)
     return msg
```

```
  File "<ipython-input-3-6bb841ea1423>", line 1
    def some_function()
                       ^
SyntaxError: invalid syntax
```

Here, Python tells us that there is a `SyntaxError` on line 1, and even puts a little arrow in the place where there is an issue. In this case the problem is that the function definition is missing a colon at the end.

Actually, the function above has *two* issues with syntax. If we fix the problem with the colon, we see that there is *also* an `IndentationError`, which means that the lines in the function definition do not all have the same indentation:

```
def some_function():
    msg = "hello, world!"
    print(msg)
     return msg
```

```
  File "<ipython-input-4-ae290e7659cb>", line 4
    return msg
    ^
IndentationError: unexpected indent
```

Both `SyntaxError` and `IndentationError` indicate a problem with the syntax of your program, but an `IndentationError` is more specific: it *always* means that there is a problem with how your code is indented.

Tabs and Spaces

A quick note on indentation errors: they can sometimes be insidious, especially if you are mixing spaces and tabs. Because they are both whitespace, it is difficult to visually tell the difference. The IPython notebook actually gives us a bit of a hint, but not all Python editors will do that. In the following example, the first two lines are using a tab for indentation, while the third line uses four spaces:

```
def some_function():
    msg = "hello, world!"
    print(msg)
    return msg
```

```
  File "<ipython-input-5-653b36fbcd41>", line 4
    return msg
              ^
IndentationError: unindent does not match any outer indentation level
```

By default, one tab is equivalent to eight spaces, so the only way to mix tabs and spaces is to make it look like this. In general, is is better to just never use tabs and always use spaces, because it can make things very confusing.

Variable Name Errors

Another very common type of error is called a `NameError`, and occurs when you try to use a variable that does not exist. For example:

```
print(a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-9d7b17ad5387> in <module>()
----> 1 print(a)

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

Variable name errors come with some of the most informative error messages, which are usually of the form "name 'the_variable_name' is not defined".

Why does this error message occur? That's harder question to answer, because it depends on what your code is supposed to do. However, there are a few very common reasons why you might have an undefined variable. The first is that you meant to use a string, but forgot to put quotes around it:

```
print(hello)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-9553ee03b645> in <module>()
----> 1 print(hello)

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

The second is that you just forgot to create the variable before using it. In the following example, `count` should have been defined (e.g., with `count = 0`) before the for loop:

```
for number in range(10):
    count = count + number
print("The count is: " + str(count))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-dd6a12d7ca5c> in <module>()
      1 for number in range(10):
----> 2     count = count + number
      3 print("The count is: " + str(count))

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

Finally, the third possibility is that you made a typo when you were writing your code. Let's say we fixed the error above by adding the line `Count = 0` before the for loop. Frustratingly, this actually does not fix the error. Remember that variables are case-sensitive, so the variable `count` is different from `Count`. We still

get the same error, because we still have not defined `count`:

```
Count = 0
for number in range(10):
    count = count + number
print("The count is: " + str(count))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-d77d40059aea> in <module>()
      1 Count = 0
      2 for number in range(10):
----> 3     count = count + number
      4 print("The count is: " + str(count))

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

Item Errors

Next up are errors having to do with containers (like lists and dictionaries) and the items within them. If you try to access an item in a list or a dictionary that does not exist, then you will get an error. This makes sense: if you asked someone what day they would like to get coffee, and they answered "caturday", you might be a bit annoyed. Python gets similarly annoyed if you try to ask it for an item that doesn't exist:

```
letters = ['a', 'b', 'c']
print("Letter #1 is " + letters[0])
print("Letter #2 is " + letters[1])
print("Letter #3 is " + letters[2])
print("Letter #4 is " + letters[3])
```

```
Letter #1 is a
Letter #2 is b
Letter #3 is c
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-d817f55b7d6c> in <module>()
      3 print("Letter #2 is " + letters[1])
      4 print("Letter #3 is " + letters[2])
----> 5 print("Letter #4 is " + letters[3])

IndexError: list index out of range
```

Here, Python is telling us that there is an `IndexError` in our code, meaning we tried to access a list index that did not exist.

File Errors

The last type of error we'll cover today are those associated with reading and writing files:

`FileNotFoundError`. If you try to read a file that does not exist, you will receive a `FileNotFoundError` telling you so.

```
file_handle = open('myfile.txt', 'r')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-14-f6e1ac4aee96> in <module>()
----> 1 file_handle = open('myfile.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
```

One reason for receiving this error is that you specified an incorrect path to the file. For example, if I am currently in a folder called `myproject`, and I have a file in `myproject/writing/myfile.txt`, but I try to just open `myfile.txt`, this will fail. The correct path would be `writing/myfile.txt`. It is also possible (like with `NameError`) that you just made a typo.

A related issue can occur if you use the "read" flag instead of the "write" flag. Python will not give you an error if you try to open a file for writing when the file does not exist. However, if you meant to open a file for reading, but accidentally opened it for writing, and then try to read from it, you will get an `UnsupportedOperation` error telling you that the file was not opened for reading:

```
file_handle = open('myfile.txt', 'w')
file_handle.read()
```

```
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
<ipython-input-15-b846479bc61f> in <module>()
      1 file_handle = open('myfile.txt', 'w')
----> 2 file_handle.read()

UnsupportedOperation: not readable
```

These are the most common errors with files, though many others exist. If you get an error that you've never seen before, searching the Internet for that error type often reveals common reasons why you might get that error.

Reading Error Messages

Read the traceback below, and identify the following pieces of information about it:

1. How many levels does the traceback have?
2. What is the file name where the error occurred?
3. What is the function name where the error occurred?
4. On which line number in this function did the error occurr?
5. What is the type of error?
6. What is the error message?

```
import errors_02
errors_02.print_friday_message()
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-e4c4cbafeeb5> in <module>()
      1 import errors_02
----> 2 errors_02.print_friday_message()

/Users/jhamrick/project/swc/novice/python/errors_02.py in print_friday_message(
)
     13
     14 def print_friday_message():
---> 15     print_message("Friday")

/Users/jhamrick/project/swc/novice/python/errors_02.py in print_message(day)
      9         "sunday": "Aw, the weekend is almost over."
     10     }
---> 11     print(messages[day])
     12
     13

KeyError: 'Friday'
```

Identifying Syntax Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. Is it a `SyntaxError` or an `IndentationError`?
3. Fix the error.
4. Repeat steps 2 and 3, until you have fixed all the errors.

```
def another_function
  print("Syntax errors are annoying.")
   print("But at least python tells us about them!")
  print("So they are usually not too hard to fix.")
```

Identifying Variable Name Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. What type of `NameError` do you think this is? In other words, is it a string with no quotes, a misspelled variable, or a variable that should have been defined but was not?
3. Fix the error.
4. Repeat steps 2 and 3, until you have fixed all the errors.

```
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (Number % 3) == 0:
        message = message + a
    else:
        message = message + "b"
print(message)
```

Identifying Item Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. What type of error is it?
3. Fix the error.

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print('My favorite season is ', seasons[4])
```