Ironhack Student Portal

preview.my.ironhack.com/lms/courses/course-

Lesson Goals

- Learn about common Python error types.
- Learn how to read Python error messages.
- Write exceptions into your code.
- Catch and handle exceptions in your code.

Introduction

A significant part of programming involves dealing with errors and exceptions. Beginner Python programmers might initially find errors annoying - constantly reminding you that you did something wrong and presenting you with obstacles in your quest for a solution that works. However, as you progress from beginner to intermediate, you will start to increasingly rely on error messages to write better code. You will learn to view errors and exceptions more as a form of communication that informs you about what is wrong with the code you are writing and where the error is occurring.

In this lesson, we will familiarize ourselves with the different types of errors and exceptions we might encounter in the Python code we write, learn how to write exceptions into our own code, and learn how to handle errors in different ways.

Python Error Types

There are many types of errors in Python, but below are a few common error types that you are likely to encounter.

- **SyntaxError:** When code has been typed incorrectly.
- **AttributeError:** When you try to access an attribute on an object that does not exist.
- **KeyError:** When you try to access a key in a dictionary that does not exist.
- **TypeError:** When an argument to a function is not of the right type (e.g. a str instead of int).
- **ValueError:** When an argument to a function is of the right type but is not in the right domain (e.g. an empty string)
- **ImportError:** When an import fails.
- **IOError:** When Python cannot access a file correctly on disk.

In addition to letting you know the type of error, Python will also provide an error message that tells you specifically why you are receiving the error.

Reading Errors

When you encounter an error in Python, there will be a Traceback that informs you of where the error occurs.

```
from pandas import does_not_exist
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-105-e9d71fb092be> in <module>()
----> 1 from pandas import does_not_exist

ImportError: cannot import name 'does_not_exist'
```

In the example above, we tried to import something that does not exist. Accordingly, Python gave us an ImportError and told us that it cannot import it. This tells us what is wrong with our code, and the part above it (where the the arrow is) shows us specifically which line of our code caused the error so that we can fix it.

This was a straightforward example, but if you get an error while using a function that itself uses other functions, you will receive several steps in the Traceback above the error line that tell you exactly which lines in which functions errored out.

```
def func(string):
    print(does_not_exist)

func('this is a string')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-116-5b48da61d415> in <module>()
      2     print(does_not_exist)
      3
----> 4 func('this is a string')

<ipython-input-116-5b48da61d415> in func(string)
      1 def func(string):
----> 2     print(does_not_exist)
      3
      4 func('this is a string')

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

In the error message above, we tried to print a string assigned to a local variable that did not exist, so we received a NameError saying that it is not defined. You can see how the first arrow tells us that there is something wrong with the `func` function and the second arrow shows us specifically that it is the line with the `print` function within `func` that is causing the issue.

Writing Exceptions Into Our Code

In addition to simply receiving errors, Python allows us to write exceptions into our code so that it throws an error when there is something wrong. We can write conditional statements into our code that check for undesirable conditions and then call the `raise` keyword when one of those conditions is met.

```
def even_number(number):
    if number % 2 != 0:
        raise ValueError("The number entered is not even!")
    else:
        print("Number accepted.")

even_number(3)
```

```
    ---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-125-195aa3301700> in <module>()
      5         print("Number accepted.")
      6
----> 7 even_number(3)

<ipython-input-125-195aa3301700> in even_number(number)
      1 def even_number(number):
      2     if number % 2 != 0:
----> 3         raise ValueError("The number entered is not even!")
      4     else:
      5         print("Number accepted.")

ValueError: The number entered is not even!
```

In the example above, we created a function that accepts even numbers. If the number passed to the function is not even, then we raise a ValueError exception and let the user know that they are receiving the error because the number they have passed is not even. The traceback shows that our `even_number` function call errored out and that it is the line with the `raise` keyword within our conditional statement that caused the error to appear.

Catching Exceptions In Our Code

The last topic we will cover in this lesson is catching exceptions in our code. Periodically, we will know that an exception may occur and our code is going to produce an error, but we will not want the error to crash our entire program. Exception handling can help us address the error while still continuing to run the rest of our code. This is done using `try` and `except` statements.

```
try:
    even_number(3)
except:
    print("The even_number function errored out.")

print("This line of code still executes.")
```

```
The even_number function errored out.
This line of code still executes.
```

In the example above, we tried to call the `even_number` function and pass it the number 3. It doesn't work, but instead of producing an error, it moves to the `except` statement, runs that code, and then continues running the rest of our code. You can use this anytime you want your code to continue running despite encountering errors.

In addition to `try` and `except`, there a couple other useful keywords used in exception handling that you should be familiar with. For example, just like in conditional statements, we can add an `else` clause to our exceptions. The `else` clause is an optional clause that can be placed after the `except` clause to execute code that should run if the `try` clause does not result in an exception.

```
try:
    even_number(4)
except:
    print("The even_number function errored out.")
else:
    print("The even number function ran successfully.")

print("This line of code still executes.")
```

```
Number accepted.
The even number function ran successfully.
This line of code still executes.
```

The other keyword you should know is `finally`. The `finally` clause is executed "on the way out" of an exception handling sequence, and it gets executed whether or not an exception has occurred. Its purpose is typically to clean up anything that was used in the sequence but won't be needed in the rest of the code.

```
try:
    even_number(4)
except:
    print("The even_number function errored out.")
else:
    print("The even number function ran successfully.")
finally:
    print("End of the sequence.")

print("This line of code still executes.")
```

```
Number accepted.
The even number function ran successfully.
End of the sequence.
This line of code still executes.
```

Nested Exceptions

Sometimes you will need to include multiple criteria in your error handling routines in order to determine whether an exception should be thrown. Just like you can nest conditional statements, you can also nest exception handling clauses to provide more specific instructions about the cause of errors to your users.

In the example below, we are defining a new `int_check` function that checks whether a number is an integer and raises a ValueError if it is not. We also start with a list of numbers and an empty `evens` list.

The nested exception handling routine within the for loop that follows will first check if each number is an integer, then check if the number is even, and then accept the input and append it to the `evens` list if it passes both exceptions. If there is an error at any step, it will print the appropriate description to the console and continue running the program.

```
def int_check(integer):
    if type(integer) != int:
        raise ValueError("The number entered is not an integer!")
    else:
        pass

numbers = [1, 2, 3, 3.7, 4, 5.5, 7, 10]
evens = []

for number in numbers:

    print("Analyzing the number:", str(number))

    try:
        int_check(number)
    except:
        print("The int_check function errored out.")
    else:
        print("The int_check function ran successfully.")
        try:
            even_number(number)
        except:
            print("The even_number function errored out.")
        else:
            evens.append(number)

    print("\n")
```

```
    Analyzing the number: 1
The int_check function ran successfully.
The even_number function errored out.


Analyzing the number: 2
The int_check function ran successfully.
Number accepted.


Analyzing the number: 3
The int_check function ran successfully.
The even_number function errored out.


Analyzing the number: 3.7
The int_check function errored out.


Analyzing the number: 4
The int_check function ran successfully.
Number accepted.


Analyzing the number: 5.5
The int_check function errored out.


Analyzing the number: 7
The int_check function ran successfully.
The even_number function errored out.


Analyzing the number: 10
The int_check function ran successfully.
Number accepted.
```

The output above shows that the numbers 1, 3, 3.7, 5.5, and 7 did not get accepted due to errors in one of the two functions. When we check our `evens` list, it should contain only the even numbers that passed all our exceptions.

```
print(evens)
```

```
[2, 4, 10]
```

Summary

In this lesson, we learned about Python errors. We began by learning some of the common error types and when they are raised. We then learned how to read Python error messages and trace back the errors we get to determine their source. We also

learned how to write code that raises errors, and about exception handling using `try`, `except`, `else`, and `finally` clauses. Finally, we got a glimpse into how you can write more complex and comprehensive exception handling into your code. You should now have a better understanding of errors you may encounter when programming with Python and how to address them.