Exceptions

by Sophia

WHAT'S COVERED

In this lesson, you will learn how to catch exceptions using the `try and except` statements. Specifically, this lesson covers:

1. Catching Exceptions Using try and except Statements

1. Catching Exceptions Using try and except Statements

We all want our programs to run perfectly all of the time. However, there are times in a real-world scenario that a program will stop. It could be due to something that the person using the program did incorrectly. Handling errors and exceptions is all about anticipating these errors and gracefully directing the user on how to fix the issue.

Let's take a look at some code where we used the input and int functions to read and convert the input, which normally returns a string to an integer number entered by the user. Can you see any possible problems that could happen with the following user input request?

EXAMPLE

```
prompt = "What is the speed of your car: "
speed = int(input(prompt))
```

The user can enter 45, 55, or even 105, which would all work as intended because we're prompting the user for an integer. But what happens if the user enters in something else, like a string? What if they entered "fast", or "105 mph"?

Consider this example, where the user asks a question instead of providing the expected input.

EXAMPLE

```
What is the speed of your car: Which car did you mean?
```

```
Trackback (most recent call last):
    File "/home/main.py", line 2, in <module>
     speed = int(input(promt))
ValueError: invalid literal for int() with base 10: 'Which car did you mean?'
```
A valid question, right? Maybe the user has several cars and wants to make sure they are answering the input request for the proper car.

Notice that when the error occurred, the script immediately stopped in its tracks with a traceback. A **traceback** is a list of functions that are executed and printed to the screen when an exception occurs. It will not execute any statement(s) that follow the exception. With the traceback that we see above, we can identify which line the error was found on, and what the error was that caused the program to stop.

Even if our code is written correctly (no syntax errors), it can error out when an attempt is made to execute it, as in our incorrect user input example above. Errors detected during execution are called **exceptions**. These exceptions are not due to a programming error. Rather, they are due to a real-world error that prevents the program from running correctly.

Here is another example, a sample program to convert a Fahrenheit temperature to a Celsius temperature:

EXAMPLE

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
```
Here is that output.

```
Enter Fahrenheit Temperature: 72
22.22222222222222
```
Entering 72 for the Fahrenheit temperature produces the equivalent Celsius temperature. Since the code converted the input to a floating-point number, we could have entered any integer, such as 35.7, -13.9, etc. They would all work correctly.

But again, what if we try to execute this code with invalid input?

```
Enter Fahrenheit Temperature: sophia
Trackback (most recent call last):
    File "/home/main.py", line 2, in <module>
     fahr = float(inp)
ValueError: could not convert string to float: 'sophia'
```
As expected, it fails.

So, how can we handle these exceptions? There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called the “try / except” statement. If you recall from an earlier lesson, `try and except` are Python keywords, meaning they cannot be used as variables. These two keywords have a primary function within code. The idea of the **try and except statement** is that if you know that some sequence of instruction(s) may have a problem, you can add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the `try and except` feature in Python as an “insurance policy” on a sequence of statements. Otherwise, the exception will terminate the program if it is left unhandled.

We can rewrite our temperature conversion code as follows:

EXAMPLE

```
inp = input('Enter Fahrenheit Temperature:')
try:
  fahr = float(inp)
  cel = (fahr - 32.0) * 5.0 / 9.0
  print(cel)
except:
  print('Oops, you did not enter in a number. Please enter a number next time.')
```

Again, let’s enter some invalid input.

```
Enter Fahrenheit Temperature: sophia
Oops, you did not enter in a number. Please enter a number next time.
```

As we'll see here, instead of receiving an unfriendly error message, we can see a message that can help us understand why there was a problem.

To break this down, Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

Handling an exception with a try statement is called *catching* an exception. In this example, the except statement prints a helpful error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

BIG IDEA

It’s important to note that catching of the exception, as we have done here, hides all of the errors, even those that may not be expected. Although this is useful in this scenario, it’s not an ideal situation, as there may be other errors that could potentially occur. We also can’t tell what the error was, specifically, either.

Let's look back at the original exception error message before we added the try except statement:

EXAMPLE

```
Enter Fahrenheit Temperature: sophia
Trackback (most recent call last):
    File "main.py", line 2, in <module>
     fahr = float(inp)
ValueError: could not convert string to float: 'sophia'
```
We can see that there was a ValueError exception. This way, if we ran into the ValueError exception, we would see our exception message.

Want to catch specific exceptions? If we know an exception based on a traceback or in the event we just want to look out for a probable exception, we can set up code to catch a specific exception. To catch a specific exception, we place the type of exception after the except keyword. Remember to finish the line off with the colon (:).

➦ EXAMPLE

```
inp = input('Enter Fahrenheit Temperature:')
try:
  fahr = float(inp)
  cel = (fahr - 32.0) * 5.0 / 9.0
  print(cel)
except ValueError:
  print('Oops, you did not enter in a number. Please enter a number next time.')
```
Now the output will “catch” a ValueError exception if the input is invalid.

```
Enter Fahrenheit Temperature: sophia
Oops, you did not enter in a number. Please enter a number next time.
```
This is useful, but we may also want to know what the actual error was from Python. We can do that as well, by setting the exception and placing it into a variable named error. This way, we can see exactly what may have gone wrong. We will need to output the variable error to see what the issue was as well.

Including the `as` keyword after the exception error allows us to set the error message and handle it accordingly. In our example below, we’ll set the variable error if there’s a ValueError exception and output it to the screen. The **as keyword** is one of Python’s reserved keywords and simply is used to create an alias. So, for our example, the `as` keyword places the error message to a variable.

➦ EXAMPLE

```
inp = input('Enter Fahrenheit Temperature:')
try:
  fahr = float(inp)
  cel = (fahr - 32.0) * 5.0 / 9.0
  print(cel)
```

```
except ValueError as error:
  print(error)
  print('Oops, you did not enter in a number. Please enter a number next time.')
```
Here is the output.

```
Enter Fahrenheit Temperature: sophia
could not convert string to float: ' sophia'
Oops, you did not enter in a number. Please enter a number next time.
```
The try statement is executed up to the point where we run into the first exception that can occur. In this case, it was the conversion of the variable `inp` to a float.

```
fahr = float(inp)
```
Notice that we tried to enter in the string “sophia” in `inp`, which the float function attempts to convert to a float value. When that happens, it goes to the except statement that’s specific to the error—which in our case is the ValueError exception. Then, we process the exception from there. This allows us to anticipate different exceptions and how the program should be able to respond to those exceptions.

TERMS TO KNOW

Traceback
A traceback is a list of functions that are executed and printed to the screen when an exception occurs.

Exceptions
Errors detected during execution (running of the code) are called exceptions.

try and except
`try and except` are reserved keywords. The idea of the `try and except` statements is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

as
The `as` keyword is one of Python’s reserved keywords and simply is used to create an alias.

SUMMARY

In this lesson, we learned about exception handling through the use of a **try and except set of statements** to **catch specific errors/exceptions**. In particular, we learned about catching a common exception called the ValueError exception, which is caused when we try to convert a variable from one data type to another. The exception occurs if the variable cannot be converted to another data type. We also learned about using the `as` keyword to set a variable to a common exception and output it to the screen during execution.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

TERMS TO KNOW

Exceptions
Errors detected during execution (running of the code) are called exceptions.

Traceback
A traceback is a list of functions that are executed and printed to the screen when an exception occurs.

as
The as keyword is one of Python’s reserved keywords and simply is used to create an alias.

try and except
try and except are reserved keywords. The idea of the try and except statements is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.