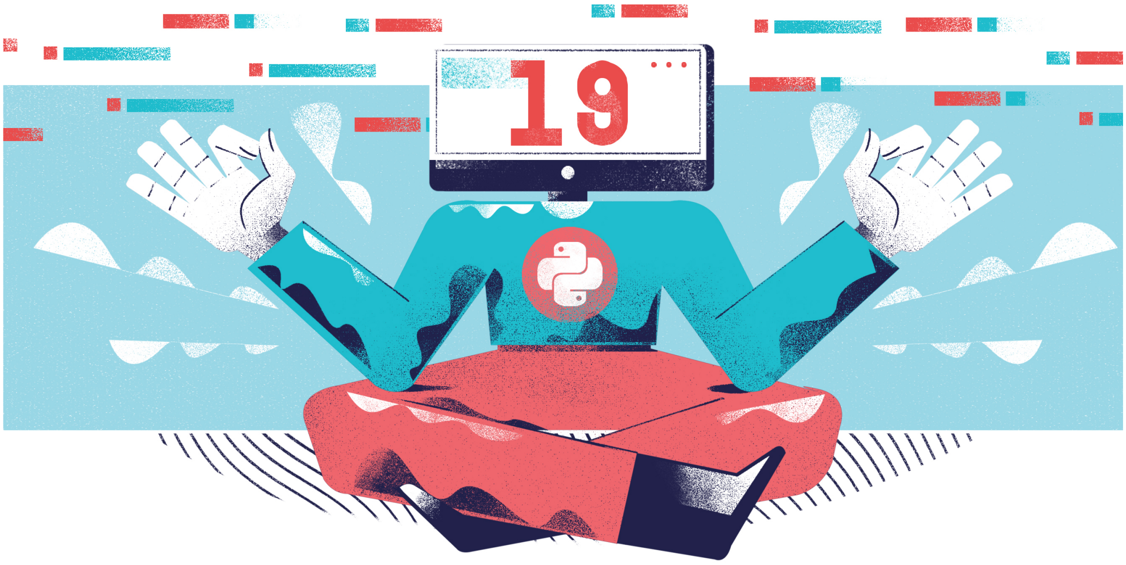
WORKING WITH MULTIPLE FILES

Day 19: Exception Handling



Welcome to day 19 of the 30 Days of Python series! Today we're going to be looking at the very important topic of exception handling.

We're also going to talk a little bit about how exceptions are used in Python, and the concept of asking for forgiveness vs. asking for permission in our code.

What are exceptions?

At this point in the series, I'm sure we've all encountered our fair share of exceptions. `SyntaxError`, `NameError`, and `TypeError` are probably starting to feel like co-workers that you see every time you sit down to code.

At the moment, encountering one of these exceptions when running our code is fatal to the application. If we try to turn user input into an

integer, and the user enters `"Five"`, the program is going to terminate, and we're going to get some lovely red text in the console about how `"Five"` isn't a base-10 integer.

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    int("Five")
ValueError: invalid literal for int() with base 10: 'F
ive'
```

Looking at an example like this, it's fair to assume that exceptions are just errors, and for the most part that assumption holds true. Most exceptions are indeed errors. However, there are some exceptions which don't really indicate that something went wrong. They're more like notifications that a given event occurred.

An example is `StopIteration`, which is probably the most common exception being raised in your applications right now: you've just never seen it, because it never terminated your programs! More on that in a little bit.

One place we see `StopIteration` is when we iterate over some iterable in a `for` loop. It's used to indicate that the all of the iterable's values have been used, and it has no new values to give us. This is how Python knows when to terminate `for` loops, and the same happens when we use destructuring to assign to several variables.

Using an exception as a signal like this is very common in Python, and this is something we're going to return to throughout this series.

Asking for permission vs asking for forgiveness

Let's return to the example from earlier where the we're trying to get an integer from the user, and they enter `"Five"` instead of the numeral we expect numeral we expect.

Our code may look something like this:

```
number = int(input("Please enter a whole number: "))
```

This type of code is nothing new to us, but it's good for us to be clear about what we're discussing.

Perhaps you disagree, but I think terminating the application in this case is a bit of a strong response to the user entering an invalid value. Even if we do want to end the program, giving the user a big red error message is probably not the best way to go. It makes it sound like something is broken, rather than there being an issue with their input. After all, we never had any plans to accept numbers written as words.

Let's think about how to solve this problem with the things we know so far.

I think a sensible approach here would be to put the prompt inside a `while` loop. We can then check if the user input is a valid integer, and if it is, we'll use that assign that value to a variable and break out of the loop. If it isn't valid, we'll move onto the next iteration, and the user will be prompted again.

The question is, how do we check if something is a valid integer value when we just have a string?

This is a bit tricky for us to do manually, but if you've been looking at the methods we have available for strings, you may have found the `isnumeric` method. We can therefore write something like this:

```
while True:
    user_number = input("Please enter a whole number: ")

    if user_number.isnumeric():
        number = int(user_number)
```

```
            break
    else:
        print("You didn't enter a valid integer!")
```

If we give it a try, it seems to work. We can enter `4` or `4834854`, and it doesn't accept `3.141592`, which is good, because would cause problems for our `int` conversion.

However, if you tested any negative numbers, you're going to uncover an issue. `isnumeric` returns `False` for negative numbers, because not all of the digits are numerals.

That's a problem, but it's not insurmountable. We can just strip off any initial `-` symbol, since we know that `int` can handle those. For this, we might use `lstrip`, rather than normal `strip`, because `lstrip` only removes the character from the left side of the string.

```
while True:
    user_number = input("Please enter a whole number: ")

    if user_number.lstrip("-").isnumeric():
        number = int(user_number)
        break
    else:
        print("You didn't enter a valid integer!")
```

Let's try again!

`-1` works no problem now, so that's great. Positive numbers still work as well, so we haven't broken anything there. However, we do have a new problem: `lstrip` is a bit too good, and it will strip off many `-` characters if it finds them. That's an issue for us, because while `-3` is a valid number as far as `int` is concerned, `--3` isn't.

That means that our if statement will accept `--3` as valid, but then converting it to an integer will give us an error.

converting it to an integer will give us an error.

If we try `--3`, we get a `ValueError`.

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    number = int(user_number)
ValueError: invalid literal for int() with base 10: '-
-3'
```

At this point, I think it's starting to become clear that we're on the wrong path. Even for this very simple case, we're having to manually deal with lots of edge cases, and it can be difficult to know if we're missing something.

This kind of approach is called "asking for permission". We're checking if something can be done in advance, and then we proceed if we determine that there aren't going to be any problems. As we've seen, this approach can be very messy, and can get extremely complicated.

This is not the approach to exception handling that we take in Python. In Python, the preferred approach is to simply attempt what we think may fail, and then to recover from an exception if one occurs. This turns the problem into a much simpler one: knowing what exceptions might occur. In the case above, we only need to worry about one exception: `ValueError`.

This alternative pattern is known as "asking for forgiveness", because we're attempting something that could go wrong, and then we're doing something to make amends if something *does* go wrong.

Let's take a look at a new piece of syntax that will allow us to use this asking for forgiveness pattern: the `try` statement.

The `try` statement

A `try` statement can get very long and detailed, but we actually only need two parts to get going. We need a `try` clause, which houses the code we expect to fail, and then usually we need at lease

one `except` clause that will describe what to do when a certain type of failure occurs.

Let's look at our number example again to see a `try` statement in action:

```
while True:
        try:
                number = int(input("Please enter a who
le number: "))
                break
        except ValueError:
                print("You didn't enter a valid intege
r!")
```

Here we have two lines of code we want to try. We would like to take in some user input and convert it to an integer, and then we would like to break out of the `while` loop if nothing goes wrong.

Our `except` clause is waiting to see if any `ValueError` is raised while we're running these operations in the `try` clause. If a `ValueError` is raised, we abandon the code in the `try` clause and we perform the actions listed in this `except` clause instead.

In this example, we simply print, `"You didn't enter a valid integer!"`, and then we run out of code in the loop body, so a new iteration of the loop begins.

Note that we don't get an error message in the console when the `ValueError` occurs.

We "handled" the exception with our `except` clause, and we provided an alternative course of action. Only unhandled exceptions terminate the application, because in those cases, we haven't provided a viable alternative. Terminating the application is really Python's last resort.

Another thing to keep in mind is that we've only handled the one

Another thing to keep in mind is that we've only handled the one type of exception here. What happens if we somehow get a `TypeError` instead?

Our `except` clause is doing nothing at all to handle a `TypeError` so we aren't providing some alternative course of action in this instance. That means a `TypeError` is still going to terminate our program, and that can sometimes be exactly what we want. Sometimes there's just nothing we can do to correct a problem, and in those cases letting an exception terminate the program is perfectly acceptable.

Important

An important thing to keep in mind is that the `try` block is going to stop running as soon as an exception occurs. If something goes wrong, it's as if none of the code in the `try` block ever ran.

Handling multiple possible exceptions

There are two ways we can handle multiple exceptions using a single `try` statement, and they have different use cases.

Let's first imagine we have two different exceptions that might occur, and we want to do different things depending on what happens. In this case we should have two `except` clauses, and each `except` clause should describe the course of action we want to take when that exception occurs.

As an example, let's create a function that calculates the mean average from some collection of numbers. We don't know what the user is going to pass into this function, so a few things can go wrong.

Here is what we're going to start off with:

```
import math

def average(numbers):
	mean = math.fsum(numbers) / len(numbers)
	print(mean)
```

```
        print(mean)
```

I'm using `fsum` here, just because the numbers are likely to be floats rather than integers in many cases.

Okay, so let's think about potential issues:

1. The user may pass in an empty collection, so then we're going to get `0` returned from `len`. That's going to lead to division by `0`, which is not allowed. In this case, we get a `ZeroDivisionError`.
2. The user may pass in something which isn't a collection. This is going to give us a `TypeError`, because `fsum` expects an iterable, and `len` expects a collection.
3. The user may pass in a collection which contains things which aren't numbers. This is also going to be a `TypeError`.

That gives us two exceptions we need to take care of: `ZeroDivisionError` and `TypeError`.

If you're not sure what kind of error is going to occur, a good approach is just to try the test case and see what happens. The documentation is also quite good at describing what exceptions occur for various operations and functions.

Now that we know what can go wrong, we can write our `try` statement.

```
import math

def average(numbers):
        try:
                mean = math.fsum(numbers) / len(number
s)
                print(mean)
        except ZeroDivisionError:
                print(0)
        except TypeError:
                print("You provided invalid values!")
```

Now we're handling situations where we get `ZeroDivisionError` differently from those where we get `TypeError` , so we're able to provide more specific feedback on what went wrong. In the case of `ZeroDivisionError` , we're not even informing the user of an issue: we've decided that the average of nothing is `0` for the purposes of our function.

If we instead want to catch both exceptions and do the same thing, we don't need two `except` clauses. We can catch multiple exceptions with the same `except` clause like so:

```
import math

def average(numbers):
        try:
                mean = math.fsum(numbers) / len(numbers)
                print(mean)
        except (ZeroDivisionError, TypeError):
                print("An average cannot be calculated for the values you provided.")
```

Important

At some point you may run across code which reads like this:

```
import math

def average(numbers):
        try:
                mean = math.fsum(numbers) / len(numbers)
                print(mean)
        except:
                print("An average cannot be calculated for the values you provided.")
```

You'll notice that we don't have any exceptions listed after the

You'll notice that we don't have any exceptions listed after the `except` clause. This is called a bare `except` clause, and it will catch any exceptions that occur.

While this has its uses, this is generally a really bad thing to use in your code. It opens up the possibility of catching many exceptions we didn't anticipate, and it can mask serious implementation problems.

The `else` clause

In addition to the `try` and `except` clauses, we can also use an `else` clause with our `try` statements. The code under the `else` clause only runs if no exceptions occur while executing the code in the `try` clause.

When finding out about `else` I know a lot of students think it's totally useless. Can't we just put more code in the `try` block?

We can, but here are a couple of reasons why this can be a really bad idea.

1. We may accidentally catch exceptions we didn't anticipate. The more code that ends up in the `try` clause, the more likely this is to happen. This may make our handling of the exception inappropriate, because we're left dealing with a situation which didn't actually occur.
2. It harms readability. The `try` clause expresses what we expect to fail, and the `except` clauses express the ways that we plan to handle specific failures in that code. The more code that gets added to the `try` clause, the less clear it is what we're actually trying, and that can make the whole structure more difficult to understand.

Does that means we always need an `else` clause? No.

Use your judgement. For very simple examples, it can be overkill, like in this example:

```python
import math

def average(numbers):
        try:
                mean = math.fsum(numbers) / len(numbers)
        except ZeroDivisionError:
                print(0)
        except TypeError:
                print("You provided invalid values!")
        else:
                print(mean)
```

We're not likely to fall prey to any issues when printing a number, and it's clear that this is not the thing we're concerned with testing.

Just make sure you don't forget about `else` when it comes to more complicated operations.

The `finally` clause

In addition to `else`, we have one more important clause available to us for `try` statements: `finally`.

`finally` is very special, because it will *always* run.

If an unhandled exception occurs, it doesn't matter. `finally` will still run its code before that exception terminates the program.

If we return from a function inside the `try` statement, `finally` will interrupt that return to run its own code first. You can see an example by running this code:

```python
def finally_flex():
        try:
                return
        finally:
                print("You return when I say you can r
```

```
                    print("You return when I say you can r
eturn...")

finally_flex()
```

This property is extremely useful for any situations where vital clean up is required after an operation. An example is when working with files. What happens if we encounter some problem while processing data in a file? We still want to close the file when we're done, and with `finally` we can make sure that this happens.

The context manager we've been using for working with files actually uses something very similar to this `finally` clause behind the scenes, to ensure that files are closed no matter what. It's so similar in fact that there are ways to make our own context managers using `try` statements with `finally` clauses.

The main use cases for `finally` are generally found in more advanced code, but it's still an important tool to know about at this stage.

Exercises

1) Create a short program that prompts the user for a list of grades separated by commas. Split the string into individual grades and use a list comprehension to convert each string to an integer. You should use a `try` statement to inform the user when the values they entered cannot be converted.

2) Investigate what happens when there is a `return` statement in both the `try` clause and `finally` clause of a `try` statement.

3) Imagine you have a file named `data.txt` with this content:

```
There is some data here!
```

Open it for reading using Python, but make sure to use a `try` block to catch an exception that arises if the file doesn't exist. Once you've verified your solution works with an actual file, delete the file and see if your `try` block is able to handle it.

When files don't exist when you try to open them, the exception raised is `FileNotFoundError` .

You can find our solutions to the exercises here.

Additional Resources

You can find more information about all of the built in exceptions here.
