ADVANCED COLLECTIONS

Day 24: Advanced Exception Handling and Raising Exceptions

Hello, and welcome to day 24 of the 30 Days of Python series! Today we're going to be expanding our exception handling toolbox and we're going to be learning how to actually raise exceptions.

We're also going to be looking into the hierarchy of exceptions which will help us be a little smarter about our exception handling.

The exception hierarchy

There are many, many different types of exception in just the standard library, and we've seen a lot of these already.

We can actually find a complete list of built in exceptions in the documentation, which is represented like so:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
```

```
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

One thing you will quickly notice is that some exceptions are listed as a subcategory of another type of exception. For example, let's take a look at `SyntaxError`.

```
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
```

We've probably all seen `SyntaxError` a fair bit, and it's there to tell us when our Python code isn't in a valid format. We've done something which violates the rules of the language.

But we can also see that there are two more exceptions that are more specific types of `SyntaxError`.

The first of these is `IndentationError`. We see this whenever we're missing some required indentation, or where there's something

missing some required indentation, or where there's something wrong with the indentation pattern we've used.

The second, `TabError`, is actually a yet more specific type of `IndentationError`, and it's used for when we have a combination of tabs and spaces for indentation. This can be problematic, because many modern editors and formatting tools are going to convert tab to spaces for you, so we could end up with different amounts of indentation than what we intended if the tabs are later converted.

It's useful to know about this hierarchy, because all of these more specific exceptions are considered to also be instances of the less specific exception they're derived from.

Let's look at another example called `LookupError`.

```
+-- LookupError
|    +-- IndexError
|    +-- KeyError
```

`LookupError` is use any time a key or index is not found, and it has two "children": `IndexError` and `KeyError`. The latter two are going to be a lot more familiar to us than `LookupError`, because those are the exceptions actually raised by things like lists, dictionaries, and tuples.

However, both `IndexError` and `KeyError` are considered to also be a `LoopkupError`, so it's perfectly legal for us to handle these exceptions like this:

```
numbers = [1, 2, 3, 4, 5]

try:
        print(numbers[100])  # <- Out of range index
except LookupError:
        print("Could not retrieve that value.")
```

Here what actually gets raised is an `IndexError`, but our `except` clause catches the exception nonetheless.

It's easy to see how this kind of pattern could be useful in cases where we have some flexible piece of code that can work with both sequences and mapping types like dictionaries.

It also gives us the ability to catch general exceptions as a fallback for more specific exceptions.

```
numbers = [1, 2, 3, 4, 5]

try:
        print(numbers[100])  # <- Out of range index
except IndexError:
        print("The requested index is out of range")
except LookupError:
        print("Could not retrieve that value.")
```

When an exception is found in a `try` clause, Python is going to look through the `except` clauses in order to see if any match. As soon as a matching exception is found, it's going to stop looking. This is very much like how conditional statements function.

In the example above, we're now going to trigger the first `except` clause, because we have an `IndexError`, but other types of `LookupError` will trigger the second `except` clause, such as a `KeyError`.

```
person = {
        "name": "Phil",
        "city": "Budapest"
}

try:
        print(person["age"])  # <- Referencing an unde
fined key
except IndexError:
        print("The requested index is out of range")
except LookupError:
        print("Could not retrieve that value.")
```

In the examples above, we're only printing something to the console, which is not a very realistic scenario, but we can replace these `print` calls with anything we like to appropriately handle different types of exception.

Accessing the original exception

Sometimes it can be useful to get hold of the original exception message, for example if we want to use it for logging purposes.

In these situations we can use the `as` keyword as part of an `except` clause, putting a variable name directly after the `as` keyword. This variable name gives us a handle that we can use to access information pertaining to the original exception.

```
numbers = [1, 2, 3, 4, 5]

try:
        print(numbers[100])  # <- Out of range index
except LookupError as ex:
        print(f"Error: {ex}")
```

Now our output looks like this:

```
Error: list index out of range
```

Note

While it may not look like it, this exception that we've caught and named `ex` is actually much more than a simple string. It contains a lot of other information, such as the original traceback, the error message, the type of the exception, etc.

We're not going to get into this in any detail here, but is useful to know about once you get a little further in your Python learning journey.

Raising an exception

When building our applications, we may encounter situations where we're unable to recover from an exception that we've tried to handle, or where we know in advance that an operation we want to perform is going to be impossible.

In situations like these, it's important that we know how to raise exceptions ourselves, so that we can terminate the program with a meaningful error message.

Raising an exception is actually very straightforward. We just need to type the word `raise` followed by the name of the exception we want to raise.

For example, we can raise a `ValueError` like this:

```
raise ValueError
```

In the console we get a traceback as usual, along with the name of the exception that caused the program to terminate.

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    raise ValueError
ValueError
```

One thing that's missing here is a message. We can add one of these easily enough by adding a set of parentheses after the exception name, and putting our message inside them.

```
raise ValueError("I raised this ValueError for no reas
on!")
```

And now the message is reflected in the console output:

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    raise ValueError("I raised this ValueError for no
 reason!")
ValueError: I raised this ValueError for no reason!
```

Reraising an exception

We can also use `raise` in another way inside an `except` clause. By writing `raise` without specifying an exception, we can reraise the exception we caught in our `except` clause.

This is very useful in cases where we don't actually want to stop an exception from happening, we just want to do something with the exception before the exception terminates the application.

Once again, logging is a good example here. We may want to use information from the original exception for our logs so that we have a permanent record of what went wrong, and then we can allow the exception to terminate the program.

Nested `try` statements

One thing you may not have realised is that we can put `try` statements inside other `try` statements. This can be very useful if we want to try to recover from some other exception, but we're not really sure if our fix is going to work.

Let's say that I'm going to be reading a very large number of numbers from a file, and I want to convert the string representation of each of these numbers to an integer. I'm fairly certain that nearly all of the numbers are going to be integers, but every now and again, I may run across a string representation of a float instead.

For simplicity's sake, lets assume that all of the numbers are on different lines in the file, so we can just iterate over the file to get what we need.

Because there's a chance I may run across a float in the file, I can't just do something like this:

```
with open("numbers.txt", "r") as numbers_file:
        numbers = [int(number) for number in numbers_f
ile]
```

If we try to pass a string representation of a float to the `int` function, for example "93.2", the program is going to terminate, because a `ValueError` will be raised by `int` .

To get around this, I'm going to define a function that is going to do the conversion for us. Inside this function I'm first going to try to convert the number to an integer using `int` , and if this fails, I'm going to try to convert it to a float instead.

Should the integer conversion go well, I'm just going to return the new integer. If we get a `float` , I'm going to round that float using the `round` function, and then I'm going to return the resulting integer.

Should all of this fail, then there's nothing more we can do, so I'm going to raise a `ValueError` with my own custom message.

```
def intify(number):
        try:
                return int(number)
        except ValueError:
                try:
                        f_number = float(number)
                except ValueError:
                        raise ValueError(f"could not c
onvert string to an integer: {number}")
                else:
                        return round(f_number)

with open("numbers.txt", "r") as numbers_file:
        numbers = [int(number) for number in numbers f
```

```
ile]
```

Controlling the traceback

One problem with the approach above is that my traceback is really big, and not very helpful:

```
Traceback (most recent call last):
  File "main.py", line 3, in intify
    return int(number)
ValueError: invalid literal for int() with base 10:
'"f"'

During handling of the above exception, another except
ion occurred:

Traceback (most recent call last):
  File "main.py", line 6, in intify
    f_number = float(number)
ValueError: could not convert string to float: '"f"'

During handling of the above exception, another except
ion occurred:

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    numbers = [intify(number) for number in numbers_fi
le]
  File "main.py", line 14, in <listcomp>
    numbers = [intify(number) for number in numbers_fi
le]
  File "main.py", line 8, in intify
    raise ValueError(f"could not convert string to an
 integer: {number}")
ValueError: could not convert string to an integer:
"f"
```

There's a lot of information here that I don't really need the user to see, because it mostly describes implementation details. I've also defined by own custom error message which explains everything the user needs to know about the situation.

To get rid of all of this extra traceback information, we can use another keyword in conjunction with `raise` called `from`.

`from` will let us specify a point from which we want to start the traceback information. We can do this by referring to an exception by name (remember that we can name exceptions using `as`). Alternatively we can write `None` in place of an exception name.

`None` is going to get rid of all of this excess traceback information from this `try` statement, and just leave us with the most recent exception.

```
def intify(number):
        try:
                return int(number)
        except ValueError:
                try:
                        f_number = float(number)
                except ValueError:
                        raise ValueError(f"could not c
onvert string to an integer: {number}") from None
                else:
                        return round(f_number)

with open("numbers.txt", "r") as numbers_file:
        numbers = [int(number) for number in numbers_f
ile]
```

Now our traceback is much more readable.

```
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    numbers = [intify(number) for number in numbers_fi
le]
  File "main.py", line 14, in <listcomp>
    numbers = [intify(number) for number in numbers_fi
le]
  File "main.py", line 8, in intify
```

```
    raise ValueError(f'could not convert string to an
integer: {number}') from None

ValueError: could not convert string to an integer:
"f"
```

Exercises

1) Ask the user for an integer between 1 and 10 (inclusive). If the number they give is outside of the specified range, raise a `ValueError` and inform them that their choice was invalid.

2) Below you'll find a `divide` function. Write exception handling so that we catch `ZeroDivisionError` exceptions, `TypeError` exceptions, and other kinds of `ArithmeticError`.

```
divide(a, b)
        print(a / b)
```

3) Below you'll find an `itemgetter` function that takes in a collection, and either a key or index. Catch any instances of `KeyError` or `IndexError`, and write the exception to a file called `log.txt`, along with the arguments that caused this issue. Once you have written to the log file, reraise the original exception.

```
def itemgetter(collection, identifier):
        return collection[identifier]
```

Project

Once you're done with today's exercises, we have another project for you! This time we're going to be creating a command line program for rolling various combinations of dice. In order to accomplish this, we're going to take in various pieces of configuration from the user before they even run the program!