Theory: User-defined exceptions

27 minutes 0 / 5 problems solved

In this topic, you are going to learn about user-defined exceptions. We will discuss the structure of custom exceptions, how to create and use them in our code.

You are familiar with how to deal with built-in exceptions. However, some programs may need user-defined exceptions for their special needs. Let's consider the following example. We need to add two integers but we do not want to work with negative integers. Python will process the addition correctly, so we can create a custom exception to raise it if any negative numbers appear. So, it is necessary to know how to work with the user-created exceptions along with the built-in ones.

§1. Raising user-defined exceptions

If we want our program to ***stop working*** when something goes wrong, we can use the `raise` keyword for the `Exception` class when a condition is met. You may come across either your or ***built-in*** exceptions like the `ZeroDivisionError` in the example below. Note that you can specify your feedback in brackets to explain the error. It will be shown when the exception occurs.

```
def example_exceptions_1(x, y):
    if y == 0:

raise ZeroDivisionError("The denominator is 0! Try again, please!")
    elif y < 0:
        raise Exception("The denominator is negative!")
    else:
        print(x / y)
```

Now let's see how this function works with different inputs:

```
example_exceptions_1(10, 0)
# Traceback (most recent call last):
#   File "main.py", line 9, in <module>
#     example_exceptions_1(10, 0)
#   File "main.py", line 3, in example_exceptions_1
#     raise ZeroDivisionError("The denominator is 0! Try again, please!")
# ZeroDivisionError: The denominator is 0! Try again, please!

example_exceptions_1(10, -2)

# Traceback (most recent call last):

#   File "main.py", line 10, in <module>

#     example_exceptions_1(10, -2)

#   File "main.py", line 5, in example_exceptions_1

#     raise Exception("The denominator is negative!")

# Exception: The denominator is negative!



example_exceptions_1(10, 5)

# 2.0
```

If there is a zero, the program will stop working and will display the built-in exception with the message you specified; note that if we had not raised this exception ourselves, it would have been raised by Python but with a regular message. If `y` is a negative integer, we get the user-defined exception and the message. If the integer is positive, it prints the results.

Current topic:

User-defined exceptions

Topic depends on:

- ✓ Elif statement 15★ (Stage 1)
- ✗ Method overriding
- ✓ Exception handling 3★

Table of contents:

§2. Creating a user-defined exception class

If you are eager to create a real code for processing user-defined exceptions in Python, you need to recall the basics of object-oriented programming. Exceptions should be derived from the `Exception` class. In the following code, we create a new class of exceptions named `NegativeResultError` derived from the built-in `Exception` class. Note that it is good to end the name of the exception class with such word as `Error` or `Exception`. We print the message informing that the procedure of creating the class is finished inside.

```
class NegativeResultError(Exception):
    print("Hooray! My first exception is working!")
```

In the `example_exceptions_2(a, b)` function below we use the `try`-`except` block. If the result of the division is positive, we just print the result. If it is negative, we *raise an exception* and go to the corresponding part of the code with `except` to print the message.

```
def example_exceptions_2(a, b):
    try:
        c = a / b
        if c < 0:
            raise NegativeResultError
        else:
            print(c)
    except NegativeResultError:
        print("There is a negative result!")
```

Let's see the results of the function for different inputs.

```
example_exceptions_2(2, 5)   # 0.4
example_exceptions_2(2, -5)  # There is a negative result!
```

Note that the message from the `NegativeResultError` class ("Hooray! My first exception is working!") is not printed anywhere, even if we raise it right away. So it makes sense to use `pass` in such simple error classes instead of printing any message.

```
raise NegativeResultError
#  Traceback (most recent call last):
#    File "main.py", line 10, in <module>
#      raise NegativeResultError
#  NegativeResultError
```

You can also create several exceptions of your own. An example is in the code below:

```
class OutOfBoundsError(Exception):
    pass

class LessThanOneError(Exception):
    pass

def example_exceptions_3(x):
    try:
        if 3 < x < 16:

            raise OutOfBoundsError

        elif x < 1:

            raise LessThanOneError

        else:

            print(x)

    except OutOfBoundsError:

        print("The value can't be between 3 and 16!")

    except LessThanOneError:

        print("The value can't be less than 1!")
```

Different errors give different outputs:

```
example_exceptions_3(2)    # 2
example_exceptions_3(5)    # The value can't be between 3 and 16!
example_exceptions_3(-10)  # The value can't be less than 1!
```

§3. Specifying exception classes

In previous sections, we displayed messages about errors by printing them ourselves in the `except`-part of the `try`-`except` block. However, we can also create the message inside the exception code using `__str__`.

```
class NotInBoundsError(Exception):
    def __str__(self):
        return "Wrong!"


def example_exceptions_4(num):
    try:
        if not 57 < num < 150:
            raise NotInBoundsError

        else:

            print(num)

    except NotInBoundsError as err:

        print(err)
```

Take a closer look at the `except {class} as {variable}` construction. It can help you print the message that is specified inside the class.

Now, the function will work as follows:

```
example_exceptions_4(46)  # Wrong!
example_exceptions_4(58)  # 58
```

What is more, if we raise the exception instead of handling it, the message will still be shown:

```
raise NotInBoundsError
# Traceback (most recent call last):
#    File "main.py", line 13, in <module>
#    raise NotInBoundsError
#  NotInBoundsError: Wrong!
```

Of course, `__str__` is not the only method for specifying your exception. The `__init__` is also suitable for working with exceptions. In the `LessThanFiveHundredError` class below, the `__init__` accepts our custom argument `num` , which is included in the message.

```
class LessThanFiveHundredError(Exception):
    def __init__(self, num):
        self.message = "The integer %s is below 500!" % str(num)
        super().__init__(self.message)


def example_exceptions_5(num):
    if num < 500:
        raise LessThanFiveHundredError(num)

    else:

        print(num)
```

The error will also show the message we specified, but now it can use the given parameters:

```
example_exceptions_5(501)
# 501
example_exceptions_5(50)
# Traceback (most recent call last):
#   File "main.py", line 15, in <module>
#     example_exceptions_5(50)
#   File "main.py", line 10, in example_exceptions_5
#     raise LessThanFiveHundredError(num)
# LessThanFiveHundredError: The integer 50 is below 500!
```

§4. Conclusion

As far as you can see, user-defined exceptions are not so tricky as they may seem at first. To deal with them, you should remember some key features:

- use the `raise` keyword to make your exception appear;
- don't forget about `Exception` class when creating your code;
- specify and customize your exception with `__init__` or `__str__`.

Only the practice counts when you try to learn something. So, switch on the following tasks to improve your skills of user-defined exceptions.

Report a typo

51 users liked this theory. **5** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion