Python Exception Handling

Summary: in this tutorial, you learn how to handle exceptions in Python in the right way by using the `try` statement.

Introduction to the exception handling in Python

To handle exceptions, you use the `try` statement. The `try` statement has the following clauses:

```
try:
    # code that you want to protect from exceptions
except <ExceptionType> as ex:
    # code that handle the exception
finally:
    # code that always execute whether the exception occurred or not
else:
    # code that excutes if try execute normally (an except clause must be present
```

Let's examine the `try` statement in greater detail.

try

In the `try` clause, you place the code that protects from one or more potential exceptions. It's a good practice to keep the code as short as possible. Often, you'll have a single statement in the `try` clause.

The `try` clause appears exactly one time in the `try` statement.

except

In the `except` clause, you place the code that handles a specific exception type. A `try` statement can have zero or more `except` clauses. Typically, each `except` clause handles different exception types in specific ways.

In an `except` clause, the `as ex` is optional. And the `<ExceptionType>` is also optional. However, if you omit the `<ExceptionType> as ex` , you'll have a bare exception handler.

When specifying exception types in the `except` clauses, you place the most specific to least specific exceptions from top to bottom.

If you have the same logic that handles different exception types, you can group them in a single `except` clause. For example:

```
try:
...
except <ExceptionType1> as ex:
    log(ex)
except <ExceptionType2> as ex:
    log(ex)
```

Become

```
try:
...
except (<ExceptionType1>, <ExceptionType2>) as ex:
    log(ex)
```

It's important to note that the `except` order matters because Python will run the first `except` clause whose exception type matches the occurred exception.

finally

The `finally` clause may appear zero or 1 time in a `try` statement. The `finally` clause always executes whether an exception occurred or not.

else

The `else` clause also appears zero or 1 time. And the `else` clause is only valid if the try statement has at least one `except` clause.

Typically, you place the code that executes if the `try` clause terminates normally.

Python exception handling example

The following defines a function that returns the result of a number by another:

```
def divide(a, b):
    return a / b
```

If you pass 0 to the second argument, you'll get a `ZeroDivisionError` exception:

```
divide(10, 0)
```

Error:

```
ZeroDivisionError: division by zero
```

To fix it, you can handle the `ZeroDivisionError` exception in the `divide()` function as follows:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        return None
```

In this example, the `divide()` function returns `None` if the `ZeroDivisionError` occurs:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        return None
```

When using the `divide()` function, you need to check if the result is `None` :

```
result = divide(10, 0)

if result is not None:
    print('result:', result)
else:
    print('Invalid inputs')
```

But returning `None` may not be the best because others may accidentally evaluate the result in the `if` statement like this:

```
result = divide(10, 0)

if result:
    print('result:', result)
else:
    print('Invalid inputs')
```

In this case, it works. However, it won't work if the first argument is zero. For example:

```
result = divide(0, 10)

if result:
    print('result:', result)
else:
    print('Invalid inputs')
```

A better approach is to raise an exception to the caller if the `ZeroDivisionError` exception occurred. For example:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        raise ValueError('The second argument (b) must not be zero')
```

In this example, the `divide()` function will raise an error if `b` is zero. To use the `divide()` function, you need to catch the `ValueError` exception:

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as ex:
        raise ValueError('The second argument (b) must not be zero')



try:
    result = divide(10, 0)
except ValueError as e:
    print(e)
```

```
else:
    print('result:', result)
```

Output:

```
The second argument (b) must not be zero
```

It's a good practice to raise an exception instead of returning `None` in special cases.

Except order example

When you catch an exception in the except clause, you need to place the exceptions from most specific to the least specific in terms of exception hierarchy.

The following shows three exception classes: `Exception` , `LookupError` , and `IndexError` :

If you catch the exception, you need to place them in the following order: IndexError, LookupErorr, and Exception.

For example, the following defines a list of three strings and attempts to access the 4th element:

```
colors = ['red', 'green', 'blue']
try:
    print(colors[3])
except IndexError as e:
    print(type(e), 'Index error')
except LookupError as e:
    print(type(e), 'Lookup error')
```

It issues the following error:

```
<class 'IndexError'> Index error
```

The `colors[3]` access causes an `IndexError` exception. However, if you swap the `except` clauses and catch the `LookupError` first and the `IndexError` second like this:

```
colors = ['red', 'green', 'blue']
try:
    print(colors[3])
except LookupError as e:
    print(type(e), 'Lookup error')
except IndexError as e:
    print(type(e), 'Index error')
```

Output:

```
<class 'IndexError'> Lookup error
```

The exception is still `IndexError` but the following message is misleading.

Bare exception handlers

When you want to catch any exception, you can use the bare exception handlers. A bare exception handler does not specify an exception type:

```
try:
    ...
except:
    ...
```

It's equivalent to the following:

```
try:
    ...
except BaseException:

    ...
```

A bare exception handler will catch any exceptions including the SystemExit and KeyboardInterupt exceptions.

A bare exception will make it harder to interrupt a program with Control-C and disguise other programs.

If you want to catch all exceptions that signal program errors, you can use except Exception instead:

```
try:
    ...
except Exception:
    ...
```

In practice, you should avoid using bare exception handlers. If you don't know exceptions to catch, just let the exception occurs and then modify the code to handle these exceptions.

To get exception information from a bare exception handler, you use the `exc_info()` function from the `sys` module.

The `sys.exc_info()` function returns a tuple that consists of three values:

- `type` is the type of the exception occurred. It's a subclass of the `BaseException` .
- `value` is the instance of the exception type.
- `traceback` is an object that encapsulates the call stack at the point where the exception originally ocurred.

The following example uses the `sys.exc_info()` function to examine the exception when a string is divided by a number:

```
import sys

try:
    '20' / 2
except:
    exc_info = sys.exc_info()
    print(exc_info)
```

Output:

```
(<class 'TypeError'>, TypeError("unsupported operand type(s) for /: 'str' and 'ir
```

The output shows that the code in the try clause causes a `TypeError` exception. Therefore, you can modify the code to handle it specifically as follows:

```
try:
    '20' / 2
except TypeError as e:
    print(e)
```

Output:

```
unsupported operand type(s) for /: 'str' and 'int'
```

Summary

- Use the `try` statement to handle exception.
- Place only minimal code that you want to protect from potential exceptions in the `try` clause.
- Handle exceptions from most specific to least specific in terms of exception types. The order of `except` clauses is important.
- The finally always executes whether the exceptions occurred or not.
- The `else` clause only executes when the `try` clause terminates normally. The `else` clause is valid only if the `try` statement has at least one `except` clause.
- Avoid using bare exception handlers.