ERRORS AND EXCEPTIONS

EXCEPTION HANDLING

An exception is an error that happens during the execution of a program. Exceptions are known to non-programmers as instances that do not conform to a general rule. The name "exception" in computer science has this meaning as well: It implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule". Exception handling is a construct in some programming languages to handle or deal with errors automatically. Many programming languages like C++, Objective-C, PHP, Java, Ruby, Python, and many others have built-in support for exception handling.

Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler. Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the programm can be continued afterwards with the previously saved data.

EXCEPTION HANDLING IN PYTHON

Exceptions handling in Python is very similar to Java. The code, which harbours the risk of an exception, is embedded in a try block. But whereas in Java exceptions are caught by catch clauses, we have statements introduced by an "except" keyword in Python. It's possible to create "custom-made" exceptions: With the raise statement it's possible to force a specified exception to occur.

Let's look at a simple example. Assuming we want to ask the user to enter an integer number. If we use a input(), the input will be a string, which we have to cast into an integer. If the input has not been a valid integer, we will generate (raise) a ValueError. We show this in the following interactive session:

```
>>> n = int(input("Please enter a number: "))
Please enter a number: 23.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
```

With the aid of exception handling, we can write robust code for reading an integer from input:

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

It's a loop, which breaks only, if a valid integer has been given.
The example script works like this:
The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left. If an exception occurs, i.e. in the casting of n, the rest of the try block will be skipped and the except clause will be executed. The raised error, in our case a ValueError, has to match one of the names after except. In our example only one, i.e. "ValueError:". After having printed the text of the print statement, the execution does another loop. It starts with a new input().

An example usage could look like this:

```
$ python integer_read.py
Please enter an integer: abc
No valid integer! Please try again ...
Please enter an integer: 42.0
No valid integer! Please try again ...
Please enter an integer: 42
Great, you successfully entered an integer!
$
```

MULTIPLE EXCEPT CLAUSES

A try statement may have more than one except clause for different exceptions. But at most one except clause will be executed.

Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:

- an IOError
- ValueError

Just in case we have an additional unnamed except clause for an unexpected error:

```
import sys

try:
```

```
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
    # e can be printed directly without using .args:
    # print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

The handling of the IOError in the previous example is of special interest. The except clause for the IOError specifies a variable "e" after the exception name (IOError). The variable "e" is bound to an exception instance with the arguments stored in instance.args.
If we call the above script with a non-existing file, we get the message:

```
I/O error(2): No such file or directory
```

And if the file integers.txt is not readable, e.g. if we don't have the permission to read it, we get the following message:

```
I/O error(13): Permission denied
```

An except clause may name more than one exception in a tuple of error names, as we see in the following example:

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred")
except:
    print("An unexpected error occurred")
    raise
```

We want to demonstrate now, what happens, if we call a function within a try block and if an exception occurs inside the function call:

```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
```

```
    print("got it :-) ", e)


print("Let's get on")
```

We learn from the above result that the function catches the exception:

```
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

We will extend our example now so that the function will catch the exception directly:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)

try:
    f()
except ValueError as e:
    print("got it :-) ", e)


print("Let's get on")
```

As we have expected, the exception will be caught inside of the function and not in the callers exception:

```
got it in the function :-)  invalid literal for int() with base
10: 'four'
Let's get on
```

We add now a "raise", which generates the ValueError again, so that the exception will be propagated to the caller:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

We will get the following result:

```
got it in the function :-)  invalid literal for int() with base
10: 'four'
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

CUSTOM-MADE EXCEPTIONS

It's possible to create Exceptions yourself:

```
>>> raise SyntaxError("Sorry, my fault!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Sorry, my fault!
```

The best or the Pythonic way to do this, consists in defining an exception class which inherits from the Exception class. You will have to go through the chapter on "Object Oriented Programming" to fully understand the following example:

```
class MyException(Exception):
    pass

raise MyException("An exception doesn't always prove the rule!")
```

If you start this program, you will get the following result:

```
$ python3 exception_eigene_klasse.py
Traceback (most recent call last):
  File "exception_eigene_klasse.py", line 4, in <module>
    raise MyException("Was falsch ist, ist falsch!")
__main__.MyException: An exception doesn't always prove the rule!
```

CLEAN-UP ACTIONS (TRY ... FINALLY)

So far the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a finally clause. Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a "finally" clause is always executed regardless if an exception occurred in a try block or not.
A simple example to demonstrate the finally clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

Let's look at the output of the previous script, if we first input a correct number and after this a string, which is raising an error:

```
bernd@venus:~/tmp$ python finally.py
Your number: 34
There may or may not have been an exception.
The inverse:  0.0294117647059
bernd@venus:~/tmp$ python finally.py
Your number: Python
There may or may not have been an exception.
Traceback (most recent call last):
  File "finally.py", line 3, in <module>
    x = float(input("Your number: "))
ValueError: invalid literal for float(): Python
bernd@venus:~/tmp$
```

COMBINING TRY, EXCEPT AND FINALLY

"finally" and "except" can be used together for the same try block, as can be seen the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

The output of the previous script, if saved as "finally2.py", for various values looks like this:

```
bernd@venus:~/tmp$ python finally2.py
Your number: 37
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: seven
```

```
You should have given either an int or a float
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: 0
Infinity
There may or may not have been an exception.
bernd@venus:~/tmp$
```

ELSE CLAUSE

The try ... except statement has an optional else clause. An else block has to be positioned after all the except clauses. An else clause will be executed if the try clause doesn't raise an exception.

The following example opens a file and reads in all the lines into a list called "text":

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
    text = fh.readlines()
    fh.close()
except IOError:
    print('cannot open', file_name)

if text:
    print(text[100])
```

This example receives the file name via a command line argument. So make sure that you call it properly: Let's assume that you saved this program as "exception_test.py". In this case, you have to call it with

```
python exception_test.py integers.txt
```

If you don't want this behaviour, just change the line "file_name = sys.argv[1]" to "file_name = 'integers.txt'".

The previous example is nearly the same as:

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
except IOError:
```

```
        print('cannot open', file_name)
    else:
        text = fh.readlines()
        fh.close()

    if text:
        print(text[100])
```

The main difference is that in the first case, all statements of the try block can lead to the same error message "cannot open ...", which is wrong, if fh.close() or fh.readlines() raise an error.