Python: Exceptions

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts of Southern Switzerland

Error management

Let's assume we have the following program, what can possibly go wrong?

```python
from math import sqrt

def perform_operation(value):
    result = sqrt(value)
    print(result)

if __name__ == '__main__':
    perform_operation(9)
    perform_operation(-9)
```

Error management

Now that we discovered a possible pitfall, we need to manage the error in some way. Do you like the proposed solution?

```python
from math import sqrt

def perform_operation(value):
    if value < 0:
        print('Error!')
        return
    result = sqrt(value)
    print(result)

if __name__ == '__main__':
    perform_operation(9)
    perform_operation(-9)
```

Error management

What about this other solution? Is it better? Pros? Cons?

```python
from math import sqrt

def perform_operation(value):
    if value < 0:
        return
    return sqrt(value)

if __name__ == '__main__':
    res = perform_operation(9)
    if res is None:
        print('Error')
    else:
        print(f'result: {res}')

    res = perform_operation(-9)
    if res is None:
        print('Error')
    else:
        print(f'result: {res}')
```

Exceptions

Exceptions are used to manage **exceptional** events that can occur during a program execution. Exceptional events can be execution errors, but also special cases, such as the inability to access certain system resources.

The support for exceptions provided by Python allows to manage these exceptional events in an explicit and consistent way, without compromising the design of the application.

Default Exception Handler

In Python, the default exception handler behaves by printing an error message to the console and by stopping the execution of the program.

It is probably something that you've already seen happening a couple of times so far.

Let's take the following simple example:

```
>>> num1 = 5
>>> num2 = 0
>>> num1 / num2
Traceback (most recent call last):
  File "/usr/lib/python3.8/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

Default Exception Handler

The same thing would have appened also if we define the code that causes the error inside a function.

This time the error also contains details about the function call.

```
>>> def function divide_numbers(num1, num2):
...     return num1 / num2

>>> divide_numbers(5, 0)
Traceback (most recent call last):
  File "/usr/lib/python3.8/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
  File "<input>", line 2, in divide_numbers
ZeroDivisionError: division by zero
```

Catching exceptions: `try` and `except`

Most of the times, the default behavior is not what we want since it will stop our program's execution.

Exceptions can be managed by means of the `try` and `except` statements.

By using the `try` statement, we can specify in its block a set of instructions for which we want to intercept possible exceptions.

The `except` statement is then used to define our own exception handler implementation. It corresponds to the `catch` keyword available in other languages.

After the `except` keyword we can specify the kind of exception we want to monitor.

The code defined inside the `except` block will be executed whenever an exception of the kind we specified occurs.

Catching exceptions: try and except

```
my_list = [1, 5]

try:
    a = my_list[3]

except IndexError:
    print('exception, went out of the allowed indexes')
```

In the above example we accessed an index of the list which is out of range.
We would usually get an `IndexError: list index out of range`, with the application stopping.

Since we executed the code in a `try` ... `except` statement, and we defined that we want to catch the exceptions of the `IndexError` kind, we will instead have our message printed, without the application stopping.

Catching exceptions: `try` and `except`

```
my_list = [1, 5]

try:
    a = my_list[3]

    # this last line will never be executed
    print(a)

except IndexError:
    print('exception, went out of the allowed indexes')

print('after catching the exception!')
```

Once the exception is handled, the program resumes after the entire `try` ... `except` blocks.

It is important to notice that it will not resume from the statement that caused the exception!

Catching multiple exceptions

If we need to catch different exceptions, for a single `try` statement, we can define multiple except blocks.

Here some examples:

```
try:
    some_operation()

except IndexError:
    # do some stuff

except NameError:
    # do some stuff

except (AttributeError, TypeError):
    # do some stuff

except:
    # do some stuff
```

Catching multiple exceptions

Other than specifying more than one except blocks with single exception, it is also allowed to specify multiple exception types using parentheses.

An empty `except` clause is also allowed. This will catch all the exceptions that haven't been yet caught by another except block.

It is anyway a bad practice to catch all of the exceptions using the empty except (or catching the Exception base class). This because there are system errors, special keyboard interrupts or simply programming mistakes that we do not want to catch.

Raising exceptions

In the previous example we have seen that it is possible that some statement containing mistakes in our code automatically cause the interpreter to launch an exception.

It is however possible to manually trigger an exception by using the `raise` keyword in combination with the exception class that we want.

```
try:
    raise IndexError

except IndexError:
    print('An exception occurred')
```

Exactly as seen before, these manually raised exceptions, if not managed properly with a custom handler, will be caught by the default exception handler.

Raising exceptions

When `raise` is called only with the exception class name, as seen in the last example, an instance of the exception is created implicitly.

This next snippet shows an equivalent way to do the same, by actually explicitly calling the constructor to create an instance of the given exception class.

Also if the instance is created before and then passed to the `raise` statement, the result would be the same.

```
try:
    raise IndexError()

except IndexError:
    print('An exception occurred')
```

Raising exceptions with arguments

So far we always raised empty exceptions, without passing any arguments to the constructor.

It is however possible to pass arguments when creating an instance of the exception object.

We could for example pass an error message, so that it will be printed out in the console together with the exception description.

```
>>> raise IndexError('My custom error message')
Traceback (most recent call last):
  File "/usr/lib/python3.8/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
IndexError: My custom error message
```

Raising exceptions with arguments

If we pass to the constructor multiple arguments, they will all be printed out and managed as a tuple.

```
>>> raise IndexError('error', 'message', 'test')
Traceback (most recent call last):
  File "/usr/lib/python3.8/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
IndexError: ('error', 'message', 'test')
```

Propagating exceptions

Sometimes it may occur that we want to catch an exception, for example for logging purposes, but we do not want to actually manage the exception.

To avoid that the exception caught dies in our exception handler, it is possible to propagate (re-raise) the current exception.

This can be done by using an empty `raise` keyword inside the exception handler.

```
try:
    raise IndexError

except IndexError:
    print('An exception occurred')
    raise
```

The `raise` statement will basically propagate the current exception to an exception handler of higher level (up in the hierarchy).

Creating custom exceptions

Instead of raising only built-in exceptions, as we did in the last example, it is also possible to define some custom Exceptions.

This allows us to create exceptions that are specific to our programs.

In order to define our own exceptions, we need to inherit from a built-in class called `Exception`.

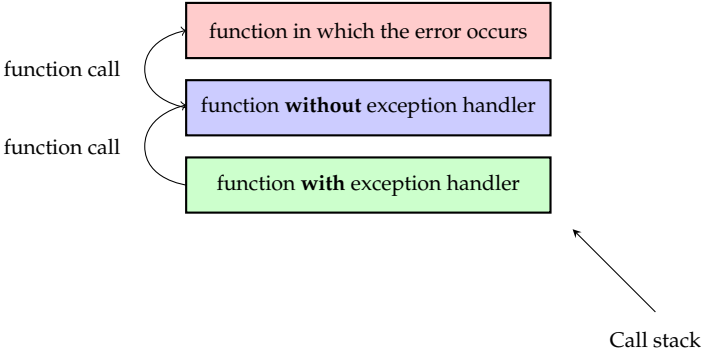
Inheritance is a very powerful mechanism from OOP that allows us to reuse code by implementing child classes.

Creating custom exceptions

```
# define our own exception as a subclass of the built-in
    Exception class
class MyException(Exception):
    pass


def exception_raiser():
    raise MyException()


if __name__ == '__main__':

    try:
        exception_raiser()

    except MyException:
        print('my exception has been caught')
```

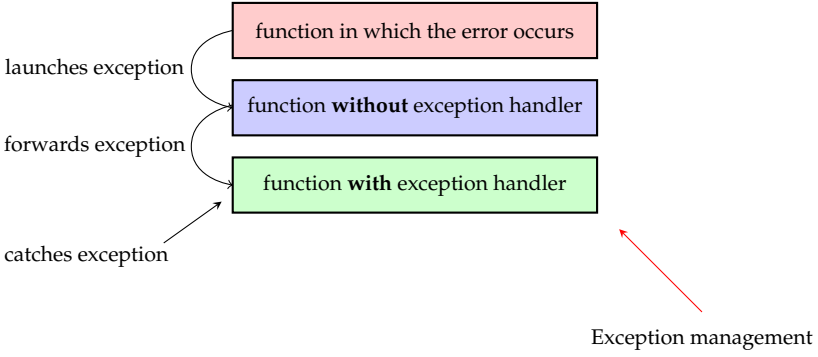
Exceptions flow

Exceptions interrupt the normal flow of the program execution:



Exceptions flow

Exceptions interrupt the normal flow of the program execution:



Exception flow: example

```
def error_launcher():
    raise IndexError

def some_function():
    # do some operations
    error_launcher()

def some_other_function():
    try:
        some_function()
    except IndexError:
        print('managing the exception')

if __name__ == '__main__':
    some_other_function()
```

Going back to the initial example...

```python
from math import sqrt

# create a class that represents the exception
class ValueBelowZeroError(Exception):
    pass

def perform_operation(value):
    if value < 0:
        # launch the exception
        raise ValueBelowZeroError()
    return sqrt(value)

if __name__ == '__main__':

    try:
        res = perform_operation(9)
        print(res)
        res = perform_operation(-9)
        print(res)
    except ValueBelowZeroError:
        # exception management
        print('Error')
```

Termination actions: `finally`

A try statement can also optionally define a `finally` block.

The `finally` block allows to specify actions that need to be carried out always at the end of a `try` . . . `except` block, no matter if an exception occurred or not.

```
try:
    perform_operation()

except MyException:
    print('Exception occurred')

finally:
    print('After perform operation')
```

The print in the finally block will be executed both if the function perform_operation causes an exception or not.

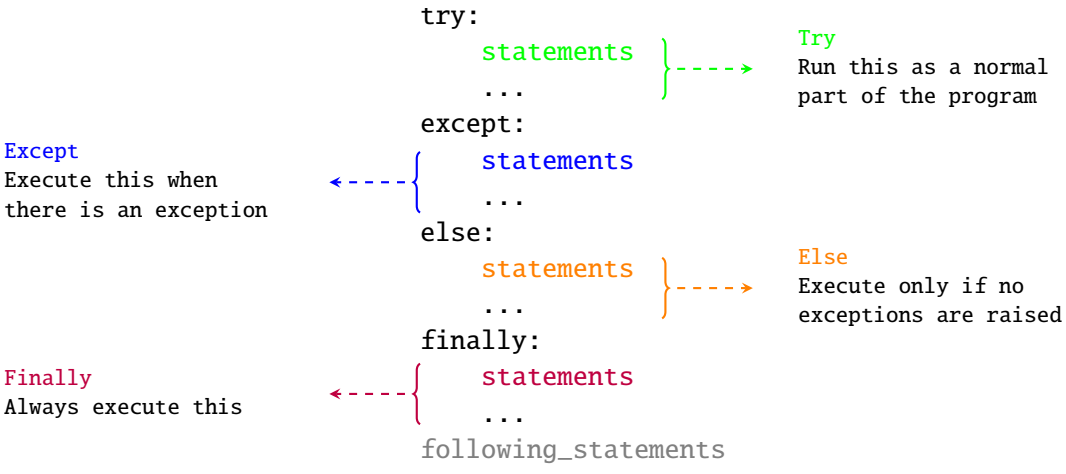
Run actions if no exception raised: `else` block

With the `try` statement it is also possible to specify an `else` block.

The `else` block allows us to specify some code that will be executed only if no exception has been raised within the `try`.

```
try:
    x = 5

except ValueError:
    print('Exception occurred')

else:
    print('Yay, no exception
        occurred!')
```

Summing up


try:
statements
...
except:
statements
...
else:
statements
...
finally:
statements
...
following_statements
Try
Run this as a normal part of the program
Except
Execute this when there is an exception
Else
Execute only if no exceptions are raised
Finally
Always execute this

Failing silently

So far we always put some sort of operations inside our `except` blocks.

In some case it could happen that we want to catch an exception in a silent way, without printing or reporting it to the user.
To do that, it is possible to use an empty placeholder, the `pass` keyword, in our except block.

```
try:
    perform_operation()

except MyException:
    pass
```

Accessing the exception instance: as

We have seen how we can raise an exception, with the possibly to pass some arguments.

When we specify our exception handler, it is also possible to have access to the instance object of the raised exception.

This is done by specifying the keyword `as` after the exception name, and providing a variable name as in the following example.

```
try:
    raise MyException()

except MyException as e:
    print(e)
```

Accessing the exception instance

From the exception instance, it is possible to print the detail about the exception or access the arguments passed in from the constructor, which are managed as a tuple.

```
try:
    raise MyException('error', 'test')

except MyException as e:
    print(e)
    # prints ('error', 'test')

    print(e.args[0])
    # prints 'error'

    print(e.args[1])
    # prints 'test'
```

Adding instance variables to custom exceptions

When using custom exceptions, since we are defining our own classes, it is also possible to specify additional data, for example in the form of instance variables.

We can achieve that by implementing our own version of the constructor method (`__init__`)

```
class MyException(Exception):
    def __init__(self, message, code):
        self.message = message
        self.code = code
```

Adding instance variables to custom exceptions

Done in this way, we can then retrieve, by accessing them from the exception instance, the values that we stored when creating the instance.

The advantage here is that we are forcing our exception to have some data passed in, which is not mandatory if using the default constructor and args approach.

```
try:
    MyException('my message', 4)
except MyException as e:
    print(e.message)
    print(e.code)
```

Adding instance variables to custom exceptions: example

```
class HttpException(Exception):
    def __init__(self, error_message, error_code):
        self.error_message = error_message
        self.error_code = error_code

def get_page(url):
    # some request for the page...
    if some_error_condition:
        raise HttpException('Request timeout', 408)
    # some other operations...

if __name__ == "__main__":
    try:
        # method that potentially raises the HttpException
        get_page("https://supsi.ch");
    except HttpException as e:
        if 400 <= e.error_code <500:
            ... # it's an error we can handle, do something
        else:
            # propagate the error
            raise e
```

Which errors to report

As the word self-explains exceptions should be used to manage exceptional events.

This means that this mechanism should not be abused. A well written program already executes all the needed checks so that it works as expected for all the use case that we can imagine.

However, sometimes we need to rely for example on external data such as user input or files, which can lead to raising exceptions.

Exception example: FileNotFound

Here another example of usage of the exceptions.

Assume that we have a function that allows us to take a file and print its content.

We can manage the situation in which the file is not existing as follows:

```
def read_file(filename):
    try:
        with open(filename) as f:
            for line in f:
                print(line)

    except FileNotFoundError:
        print(f'File {filename} not existing')
```

Summary

- Exceptions
- Catching exceptions: `except`
- Raising exceptions: `raise`
- Custom exceptions: `Exception` subclasses
- Exceptions flow
- Termination actions: `finally, else`
- Failing silently: `pass`
- Exceptions instance: `as`, `args` and instance variables

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 33-36
- Python Crash Course, no starch press - Eric Matthes: Chapter 10
- Python Official Documentation: https://docs.python.org/3/tutorial/classes.html
- LearnByExample: https://www.learnbyexample.org/python/