Introduction

Some run time errors can be anticipated. They must be dealt with using dedicated code instructions: that is called **exceptions handling**.

An exception is a Python object that tells the user about an error occuring at a specific instruction. These exceptions are of several types since they describe several different problems: type errors (`TypeError`), index errors (`IndexError`), ...

In [1]:

```
s = "a string"
s / 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 s = "a string"
----> 2 s / 5

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

In [2]:

```
var = [0, 1, 2, 3]
var[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[2], line 2
      1 var = [0, 1, 2, 3]
----> 2 var[4]

IndexError: list index out of range
```

A message exhibits the problematic instruction in what is called a `Traceback`. A `Traceback` is a list of code instructions concerned by the exception, going from most recent call to a function (the problematic one) to oldest call.

In [3]:
```
def f1(a, b):
    return a / b

def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)
```

In [4]:
```
f3(1, 0)
print("Not executed")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[4], line 1
----> 1 f3(1, 0)
      2 print("Not executed")

Cell In[3], line 8, in f3(a, b)
      7 def f3(a, b):
----> 8     f2(a, b)

Cell In[3], line 5, in f2(a, b)
      4 def f2(a, b):
----> 5     f1(a, b)

Cell In[3], line 2, in f1(a, b)
      1 def f1(a, b):
----> 2     return a / b

ZeroDivisionError: division by zero
```

Handle an exception

If nothing particular is done, an exception is **blocking** for the running code and the process is terminated. To prevent this termination, one must use a `try` code block:

If an instruction fails in `try`, an exception is raised (as usual) but is not blocking: the content of `except` is ran instead:

In [5]:

```
def f1(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        print("`b` was 0, met the following exception: ", e)


def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)

f3(1, 0)
print("Executed")
```

```
`b` was 0, met the following exception:  division by zero
Executed
```

Notes:

- **Whenever it's possible**, one must specify the type of exception to **'catch'** (here `ZeroDivisionError`). Yet, it is also possible to only write `except:` in order to catch all types of exceptions.
- The syntax `as e` store the exception instance (an instance of type `ZeroDivisionError`) in variable `e`.
- Several `except` can follow each other to catch different types of exceptions or define several exceptions types in one `except` (see examples below).

In [6]:

```
def f1(a, b):
    try:
        a / b
        b[5]
    except ZeroDivisionError as e:
        print("`b` was 0, met the following exception: ", e)
    except TypeError as e:
        print(f"`b` was of type {type(b)}, met the following exception: ", e)


def f2(a, b):
    f1(a, b)

def f3(a, b):
    f2(a, b)
```

In [7]:
```
f3(1, 0)
f3(0, 1)
```

```
`b` was 0, met the following exception:  division by zero
`b` was of type <class 'int'>, met the following exception:  'int' object is
not subscriptable
```

In [8]:
```
def f1(a, b):
    try:
        a / b
        b[5]
    except (ZeroDivisionError, TypeError) as e:
        print(e)
f1(0, 1)
```

```
'int' object is not subscriptable
```

The `try` code section must include as few instructions as possible (so that unpredicted errors won't be covered by `except`).

Thus, the `else` section is used: instructions in `else` are executed if risky code in `try` runs with no exception. Put differently: `else` is not ran if `except` is ran.

In [9]:
```
def f1(a, b):
    try:
        x = a / b
    except (ZeroDivisionError, TypeError) as e:
        print(e)
    else:
        y = 2 * x + 5
        return y
f1(5, 2)
```

Out[9]:
```
10.0
```

Note: another clause exists: `finally`. Instructions in `finally` will be executed just before `try` terminates (i.e. before an exception is raised within `try`, or after the very last instruction of `try`). It is useful for some advanced cases.

Raise an exception

It is possible to create a code interruption depending on certain conditions. In this case, with use the `raise` statement. In the example below, a `ValueError` is raised whenever the acquired value is negative. This error is catched in using a dedicated `except`.

In [10]:

```python
from random import randint

def sensor_reading(only_valid_data=True):
    # fake real-time data acquisition
    new_value = randint(-1, 10)
    if new_value < 0 and only_valid_data:
        raise ValueError(f"Sensor default, got negative value {new_value}.")
    return new_value


def data_acquisition(replacement_value):
    data = []
    for k in range(20):
        try:
            new_value = sensor_reading()
        except ValueError as e:
            print(f"Time step {k}: ", e)
            new_value = replacement_value
        data.append(new_value)
    return data

data_acquisition(0)
```

Out[10]:

```
[10, 4, 2, 8, 1, 6, 1, 9, 2, 0, 1, 6, 0, 4, 0, 2, 3, 10, 6, 7]
```

Conclusion

`try` / `except` is a powerful way to handle expected errors, i.e. errors that will likely happen and that need a special treatment.