> Python workshop
>
> ***3-4. Epilogue***

What's next?

- Remember that Python is only one of the tools you could use.
- We hope that by learning Python, you understand how to solve problems through coding.
- Once you gain the experience in one programming language, it should be easy for you to learn the other.

Other programming languages to learn

- **C++**: for its traditional structural, procedural approach.
- **C**: for its low-level memory managemenet.
- **Java**: for its strongly typed characteristics, also for its Object-oriented Programming design.
- **JavaScript**: for its asynchronous feature, also for its functional programming support.

Self-learning topics

There are no quizzes and exercises for this section. You may come back to this section later.

Other topics in Python

- Errors and Exceptions
- File I/O

Errors and Exceptions

During the course of the workshop, you should have already encountered a lot of errors. There are mainly two types of errors, **syntax error** and **runtime error**.

Syntax errors

Syntax errors are errors in the syntax, codes cannot be executed if the code is incorrect in syntax. For example, the following code is missing a colon.

In [1]:
```
a = 1
if a < 0
    print(a, 'is negative')
```

```
  Cell In[1], line 2
    if a < 0
           ^
SyntaxError: expected ':'
```

Runtime error

Runtime errors are errors happens when code is executed, for example when you divide a value by zero, trying access a variable that is not defined, etc.

In [2]:
```
a = 1
print(a/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[2], line 2
      1 a = 1
----> 2 print(a/0)

ZeroDivisionError: division by zero
```

Exception handling

We can wrap our code with `try...except` to capture **runtime errors** and handle them:

In [3]:
```
try:
    a = 0
    print(1/a)
except:
    print('What have you done?')
```

```
What have you done?
```

Specific exception handling

You can specify the kind of errors to be captured. Try to enter `0` or some characters in the example below.

In [4]:
```
try:
    a = int(input())
    print(1/a)
except ZeroDivisionError:
    print('Cannot divide by zero!')
except:
    print('What have you done?')
```

```
 0
Cannot divide by zero!
```

The last `except` capture all other errors that is not captured.

You can also capture multiple errors with the same `except`.

In [5]:
```python
try:
    a = int(input())
    print(1/a)
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
```

```
 0
I need a number but no zero please!
```

Exception info

We can assign a variable for the exception caught, in order to collect information from it.

In [6]:
```python
try:
    a = int(input())
    print(1/a)
except (ZeroDivisionError, ValueError) as err:
    print('Error captured:', err)
    print('I need a number but no zero please!')
```

```
 0
Error captured: division by zero
I need a number but no zero please!
```

Raise exception

We can raise an exception ourselves using `raise`. Try inputing `1` in the example below.

In [7]:
```python
try:
    a = int(input())
    if a == 1:
        raise ValueError("I don't like 1.")
except ValueError as err:
    print('Error captured:', err)
```

```
 1
Error captured: I don't like 1.
```

Custom exception

We can define our own error by extending the Exception class.

In [8]:
```python
class MyException(Exception):
    pass
```

```
try:
    a = int(input())
    if a == 1:
        raise MyException("I don't like 1.")
except MyException as err:
    print('Error captured:', err)
```

```
 1
Error captured: I don't like 1.
```

else

We can add an `else` clause at the end, which will be executed when the `try` block finished successfully without catching an exception. This is useful in presenting a result after `try-catch`.

In [9]:
```
try:
    a = int(input())
    b = 1/a
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
else:
    print('1 over', a, 'is', b)
```

```
 1
1 over 1 is 1.0
```

finally

We can add an `finally` clause at the end, which will always be executed at the end. This is useful for clean up purpose, for example if we opened a file or a network connection in `try`, we can clean them up in `finally`.

In [10]:
```
try:
    a = int(input())
    b = 1/a
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
else:
    print('1 over', a, 'is', b)
finally:
    print('done.')
```

```
 1
1 over 1 is 1.0
done.
```

File I/O

File I/O usually requires exception handling and therefore it is introduced here at the end.

Note that you need to try the examples below in your own environment.

Open a file

To open a file, we can use the `open()` function. A file name and a mode should be specified.

```
f = open('test.txt', 'r')
f.close()
```

- `r` is the mode of accessing the file. This can be `r` for reading, `w` for writing, and `r+` for both.
- File must be closed with `close()` after accessing.

Using `with`

We can also use a `with` block for file access, file will be automatically closed in this case:

```
with open('test.txt', 'r') as f:
    pass
```

This is the preferred way of accessing a file.

File reading

To read the entire file, we can use `read()`:

```
with open('test.txt', 'r') as f:
    s = f.read()
    print(s)
```

Alternatively we can read one line instead using `readline()`:

```
with open('test.txt', 'r') as f:
    line = f.readline()
    print(line)
```

Iterating a file object

In fact, the file object can be iterated. In that case, the file is read line by line.

```
with open('test.txt', 'r') as f:
    for line in f:
        print(line)
```

Writing to file

- The `write()` function write contents to a file.
- Note that opening a file in `w` mode will overwrite the whole file. You should use `r+` mode if you are only modifying a file.

```
with open('test.txt', 'w') as f:
    f.write("Hello, world")
```

Seeking

- The `seek()` function moves the current read/write position of a file.
- It takes 2 arguments, the first one is the movement, the second one is the starting point of movement, with `0` being the beginning of the file, `1` being the current position, and `2` being the end of file.
- For example we can append to a file like this:

```
with open('test.txt', 'r+') as f:
    f.seek(0, 2)
    f.write("Hello, world")
```