ERRORS AND EXCEPTIONS

No matter your skill as a programmer, you will eventually make a coding mistake. Such mistakes come in three basic flavours:

- `Syntax errors`: Errors where the code is not valid Python (generally easy to fix)
- `Runtime errors`: Errors where syntactically valid code fails to execute, perhaps due to invalid user input (sometimes easy to fix)
- `Semantic errors`: Errors in logic: code executes without a problem, but the result is not what you expect (often very difficult to track-down and fix)

Here we're going to focus on how to deal cleanly with `runtime errors`. As we'll see, Python handles runtime errors via its **exception handling** framework.

RUNTIME ERRORS

You might have already come across `runtime errors` in the previous notebooks. They can happen in a lot of ways. For example, if you try to reference an *undefined variable*:

```
print(Q)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-cbf1bd89097d> in <module>
----> 1 print(Q)

NameError: name 'Q' is not defined
```

Or if you try an operation that's not defined:

```
1 + 'abc'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a51a3635a212> in <module>
----> 1 1 + 'abc'

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

Or you might be trying to compute a mathematically ill-defined result:

```
2 / 0
---------------------------------------------------------------------------
```

```
ZeroDivisionError                          Traceback (most recent call
last)
<ipython-input-3-8b4ac6d3a3e1> in <module>
----> 1 2 / 0

ZeroDivisionError: division by zero
```

Or maybe you're trying to access a sequence element that doesn't exist:

```
L = [1, 2, 3]
L[1000]
----------------------------------------------------------------------
-
IndexError                              Traceback (most recent call
last)
<ipython-input-4-354067ebdc84> in <module>
      1 L = [1, 2, 3]
----> 2 L[1000]

IndexError: list index out of range
```

Note

Note that in each case, Python is kind enough to not simply indicate that an error happened, but to spit out a *meaningful* exception that includes information about what exactly went wrong, along with the exact line of code where the error happened. Having access to meaningful errors like this is immensely useful when trying to trace the root of problems in your code.

TRY AND EXCEPT BLOCK

The main tool Python gives you for handling runtime exceptions is the `try...except` clause. Its basic structure is this:

```
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")
```

Note

The second block here did not get executed: this is because the first block did not return an error.

Let's put a problematic statement in the `try` block and see what happens.

```python
try:
    print("let's try something:")
    x = 1 / 0 # ZeroDivisionError
except:
    print("something bad happened!")
let's try something:
something bad happened!
```

Here we see that when the error was raised in the `try` statement (in this case, a `ZeroDivisionError`), the error was caught, and the `except` statement was executed.

One way this is often used is to check user input within a function or another piece of code. For example, we might wish to have a function that catches zero-division and returns some other value, perhaps a suitably large number like 1010:

```python
def safe_divide(a, b):
    try:
        return a / b
    except:
        return 10**10
safe_divide(1, 2)
0.5
safe_divide(2, 0)
10000000000
```

There is a subtle problem with this code, though: what happens when another type of exception comes up? For example, this is probably not what we intended:

```python
safe_divide (1, '2')
1e+100
```

Dividing an integer and a string raises a `TypeError`, which our over-zealous code caught and assumed was a `ZeroDivisionError`! For this reason, it's nearly always a better idea to catch exceptions *explicitly*:

```python
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return 10**10
safe_divide(1, 0)
10000000000
safe_divide(1, '2')
----------------------------------------------------------------------
-
```

```
TypeError                                    Traceback (most recent call
last)
<ipython-input-9-cbb3eb91a66d> in <module>
----> 1 safe_divide(1, '2')

<ipython-input-7-897e35888e3b> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 10**10

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

We're now catching zero-division errors only, and letting all other errors pass through un-modified.

RAISE STATEMENT

We've seen how valuable it is to have informative exceptions when using parts of the Python language. It's equally valuable to make use of informative exceptions within the code you write, so that users of your code (foremost yourself!) can figure out what caused their errors.

The way you raise your own exceptions is with the `raise` statement. For example:

```
raise RuntimeError("my error message")
---------------------------------------------------------------------
-
RuntimeError                                 Traceback (most recent call
last)
<ipython-input-14-b1834d213d3b> in <module>
----> 1 raise RuntimeError("my error message")

RuntimeError: my error message
```

DIVING DEEPER INTO EXCEPTIONS

Briefly, we want to mention here some other concepts you might run into. We'll not go into detail on these concepts and how and why to use them, but instead simply show you the syntax so you can explore more on your own.

ACCESSING THE ERROR MESSAGE

Sometimes in a `try...except` statement, you would like to be able to work with the error message itself. This can be done with the `as` keyword:

```python
try:
    x = 1 / 0
except ZeroDivisionError as err:
    print("Error class is:  ", type(err))
    print("Error message is:", err)
Error class is:   <class 'ZeroDivisionError'>
Error message is: division by zero
```

TRY...EXCEPT...ELSE...FINALLY

In addition to `try` and `except`, you can use the `else` and `finally` keywords to further tune your code's handling of exceptions. The basic structure is this:

```python
try:
    print("try something here")
except:
    print("this happens only if it fails")
else:
    print("this happens only if it succeeds")
finally:
    print("this happens no matter what")
try something here
this happens only if it succeeds
this happens no matter what
```

The utility of `else` here is clear, but what's the point of `finally`? Well, the `finally` clause really is executed *no matter what*: it is generally used to do some sort of cleanup after an operation completes. For example, closing the file or database connection.