9 Exceptions

Python uses special objects, called **exceptions** to react to errors that arise during a program run. Whenever an error occurs and Python interpreter is unsure what should be done next, it creates and exception object. If a programmer wants the program to continue its execution, the code that handles the exception (**exception handler**) should be written.

Exceptions in Python are usually handled with `try-except` blocks. A `try-except` block asks Python to do something (that can possibly end with an error) but also tells the interpreter what to do, when an error occurs (**exception is raised**). When `try-except` blocks are used, the program will continue, provided that all the errors were accounted for. This is the method to produce more resilient code.

It is a good practice to add an exception-handling code anywhere an exception might occur. If an exception is occurs in some function and is not **caught**, the function run is interrupted and the code that called the function is searched in order to find a matching exception handler, until *all the way up* is checked. If the exception is not caught anywhere along the way, Python prints an error message and usually some additional information (making it easier to resolve the issue in the next run) and terminates the program (example `[1]`).

In [1]:
```
list_3 = [1, 3, 4]
print(list_3[7]) # use non-existing index
```

```
IndexError                                Traceback (most recent call last)
<ipython-input-1-2a2e2d721a75> in <module>
1 list_3 = [1, 3, 4]
----> 2 print(list_3[7]) # use non-existing index

IndexError: list index out of range
```

9.1 Using `try--except` blocks

Generally, when an error occurs during a program run, an exception is raised and the program stops. To catch the errors (exceptions) earlier and avoid run interruption, a `try--except` blocks are used.

All the errors that you have previously seen in traceback, e.g. `ZeroDivisionError`, `IndexError`, `KeyError`, `ValueError` etc., are `exception` objects. The information on the kind of error can be used to modify the program run when it occurs.

To prepare for the errors, write a `try--except` block to handle the exception that might be raised (example `[2]`). If the code in the `try` block works (runs without any errors), the `except` block is skipped. If the instructions in `try` block cause an error, Python looks for an `except` block that matches the occurring error and runs the code inside the `except` block.

In [2]:
```
try:
    # operation that can go wrong
    z = 5 / 0
except ZeroDivisionError:
    # what should be done when this particular error appears
    print("You can't divide by zero!")
```

```
You can't divide by zero!
```

It is possible for errors to **fail silently** – instead of a comment or action in the **except** block, a `pass` can be typed. In case of an error a program will continue as if nothing happened. However, it is not

encouraged to do so, in fact, one of the lines of the Zen of Python strongly advises against silencing errors[1]:

> Errors should never pass silently. Unless explicitly silenced.

It is up to the programmer how much information about the program run is shared with the user.

If an error is raised and it is not listed in the `except` clause, the error is propagated "up", to look for another `except` clause. "Up" in this case means "an indentation level less" (or back to the code which called the code where error occured). It is not easy to predict all the errors and check every possibility. Sometimes it is easier to execute the code and deal with errors when they happen with a very general `try-except` block (example [3]).

In [3]:
```
try:
    # do something which may end with an error
    file = open('non-existing')
except:
    #do something if error appers
    print("Oh no, something went wrong"))
```

```
Oh no, something went wrong
```

9.2 The `else` block

So far we have covered the case in which an error occurs in our program and we want to continue the run. However, is there any way to differentiate the script behaviour between the successful and erroneous run? Yes! It requires adding one more block to the `try-except` part of the program, an `else` block.

The code included in the `else` block is executed only when **no exception is raised**. The `else` block should contain any code that depends on the `try` block **succeeding** (example [4]).

In [4]:
```
try:
    # operation that can go wrong
    z = 5 / 0
except ZeroDivisionError:
    # what should be done when this particular error appears
    print("You can't divide by zero!")
else:
    # in case of try block being successul
    print(z)
```

```
You can't divide by zero!
```

9.3 The `finally` block

To include some block of code to do some clean-up in case an error occurs, a `finally` block can be added to the usual exception-handling `try-except` blocks (example [5]).

[1] To read full *Zen of Python* try running a command `import this`

In [5]:
```
try:
    file = open("f.txt", 'r') # file might not exist, file.close() will raise another
        error
    # operation that can go wrong
    data = file.read () # may raise UnicodeDecodeError
except: # any error
    # what should be done when this particular error appears
    print("Error reading file")
finally:
    # always runs after try
    file.close() # safely closing a file
    print("file closed")
```

9.4 Raising exceptions

Exceptions can be raised also by the programmer. Raising an exception is also referred as **throwing an exception**, contrary to handling the exception: **catching the exception** (examples `[6]`, `[7]`). We alarm the interpreter that the error occurs using `raise` command.

In [6]:
```
raise KeyError("Message")
```

```
KeyError                                Traceback (most recent call last)
<ipython-input-16-e8f85212de20> in <module>
----> 1 raise KeyError("Message")

KeyError: 'Message'
```

In [7]:
```
e = KeyError("Error message")
raise e
```

```
KeyError                                Traceback (most recent call last)
<ipython-input-15-c76cef722649> in <module>
1 e = KeyError("Error message")
----> 2 raise e

KeyError: 'Error message'
```

9.5 Creating own exceptions

New errors can be defined based on the `Exception` class and if the unwanted behaviour appears, they can be raised with a `raise` statement (example `[8]`).

In [8]:
```
class MyException(Exception):
    pass # the exception class does not need any content

raise MyException("Raised my own exception!")
```

```
MyException                               Traceback (most recent call last)
<ipython-input-3-44bf03c7e47f> in <module>
2    pass
3
----> 4 raise MyException("Raised my own exception!")

MyException: Raised my own exception!
```

9.6 LBYL vs EAFP

Two approaches are possible when it comes to avoiding errors: LBYL and EAFP.

EAFP, which stands for *Easier to Ask Forgiveness than Permission*, is the Pythonic way. This coding style assumes the existence of valid keys or attributes and catches the exceptions if the assumptions prove false. It is characterized by `try` and `except` statements [2].

In [9]:
```
lst:eafp]
numerator = 8
denominator = 0

try:
    print(numerator / denominator)
except ZeroDivisionError:
    print("Division by zero not possible!")
```

EAFP contrasts with another technique, LBYL, short for *Look Before You Leap* which assumes checking for pre-conditions explicitly, before making calls or lookups. It is characterized by presence of many `if` statements (example `[10]`). Using this approach may cause problems in a multi-threaded environment, that is why the EAFP is recommended.

In [10]:
```
numerator = 8
denominator = 0

if denominator == 0:
    print("Division by zero not possible!")
else:
    print(numerator / denominator)
```

9.7 Exercises

Exercise 9.1. Check out the list of exceptions in Python (`https://www.tutorialspoint.com/python3/standard_exceptions.htm`). Try to write a few pieces of code that raises three of those exceptions. Handle the exceptions using `try-except`.

Exercise 9.2. Take a look at the code listed below. Add 'try-except' statement where necessary to avoid crashing the program.

[2] `http://docs.python.org/glossary.html#term-eafp`

In [11]:
```
def example1():
       for i in range(3):
           x = int(input("enter a number: "))
           y = int( input("enter another number: "))
           print( x, '/', y, '=', x/y )

def example2(L):
       product = 0
       product_of_pairs = []
       for i in range(len(L)):
           product_of_pairs.append(L[i]+L[i+1])
       print( "product_of_pairs = ", product_of_pairs )


def printUpperFile(fileName ):
       file = open(fileName, "r")
       for line in file:
       print(line.upper())
       file.close()

def main():
       example1()
       L = [10, 3, 5, 6, 9, 3]
       example2(L)
       example2([10, 3, 5, 6, "NA", 3])
       printUpperFile("notexisting.txt" )

main()
```

9.8 Useful links

- Built-in exceptions: `https://docs.python.org/3/library/exceptions.html#bltin-exceptions`
- Exceptions (Python tutorial): `https://docs.python.org/3/tutorial/errors.html`
- Exceptions: `https://www.tutorialspoint.com/python/python_exceptions.htm`

References

Beazley, D. M. (2009). *Python essential reference*. Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book*. Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].