PROGRAMMING FUNDAMENTALS

EXCEPTIONS

João Correia Lopes

INESC TEC, FEUP

18 december 2018

GOALS

By the end of this class, the student should be able to:

- Write code to catch and handle *runtime* exceptions that may occur during program execution
- Raise exceptions when a program detects an error condition
- Assert conditions that must be true during execution, orelse throw an error

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Appendix E) [PDF]
- The Python Tutorial, *8. Exceptions*, Python 3.6.7 documentation, Release 3.6.7, November 20, 2018 [HTML]

TIPS

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CODE, TEST & PLAY

- Have a look at the code in GitHub: **https://github.com/fpro-admin/lectures/**
- Test before you submit at FPROtest: **http://fpro.fe.up.pt/test/**
- Pay a visit to the playground at FPROplay: **http://fpro.fe.up.pt/play/**

CONTENTS

1 EXCEPTIONS

- E.1 Catching exceptions
- E.2 Raising our own exceptions
- E.3 Revisiting an earlier example
- E.4 The `finally` clause of the `try` statement
- The `assert` statement
- Examples & Summary

SOME COMMON EXCEPTIONS

Here are some basic exceptions that you might encounter when writing programs:

- `NameError` — raised when the program cannot find a local or global name
- `TypeError` — raised when a function is passed an object of the inappropriate type as its argument
- `ValueError` — occurs when a function argument has the right type but an inappropriate value
- `ZeroDivisionError` — raised when you provide the second argument for a division or modulo operation as zero
- `FileNotFoundError` — raised when the file or directory that the program requested does not exist

⇒ https://code.tutsplus.com/tutorials/

Runtime errors

- Whenever a runtime error occurs, it creates an **exception** object
- The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred
- The error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon

```
>>> tup = ("a", "b", "d", "d")
>>> tup[2] = "c"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: ’tuple’ object does not support item assignment
```

CATCHING EXCEPTIONS

- Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop
- We can handle the exception using the `try` statement to “wrap” a region of code

```python
filename = input("Enter a file name: ")
try:
    f = open(filename, "r")
except FileNotFoundError:
    print("There is no file named", filename)
```

- A `else` block is executed after the `try` one, if no exception occurred
- A `finally` block is executed in any case

⇒ https://github.com/fpro-admin/lectures/blob/master/23/try.py

CATCHING EXCEPTIONS

- Sometimes we want to execute an operation that might cause an exception, but we don’t want the program to stop
- We can handle the exception using the `try` statement to “wrap” a region of code

```
filename = input("Enter a file name: ")
try:
    f = open(filename, "r")
except FileNotFoundError:
    print("There is no file named", filename)
```

- A `else` block is executed after the `try` one, if no exception occurred
- A `finally` block is executed in any case

⇒ https://github.com/fpro-admin/lectures/blob/master/23/try.py

RAISING OUR OWN EXCEPTIONS

- Can our program deliberately cause its own exceptions?
- If our program detects an error condition, we can raise an exception
- If there's a chain of calls, "*unwinding the call stack*" takes place until a `try ... except` is found

```
def get_age():
    age = int(input("Please enter your age: "))
    if age < 0:
        # Create a new instance of an exception
        my_error = ValueError("{0} is not a valid age".format(age))
        raise my_error
    return age
```

⇒ https://github.com/fpro-admin/lectures/blob/master/23/age.py

REVISITING AN EARLIER EXAMPLE

- Using exception handling, we can now modify our `recursion_depth` example from the previous chapter so that it stops when it reaches the maximum recursion depth allowed

⇒ `https://github.com/fpro-admin/lectures/blob/master/23/rec_depth.py`

FINALLY

- A common programming pattern is to grab a resource of some kind
- Then we perform some computation which may raise an exception, or may work without any problems
- Whatever happens, we want to “clean up” the resources we grabbed

⇒ https://github.com/fpro-admin/lectures/blob/master/23/show_poly.py

ASSERTIONS

- Assertions are statements that assert or state a fact
- Assertions are simply boolean expressions that checks if the conditions return true or not: if it's false, the program stops and throws an error
- `assert` statement takes an expression and optional message
- Assertions are used to check types, values of argument and the output of the function
- Assertions are used as debugging tool as it halts the program at the point where an error occurs

⇒ https://github.com/fpro-admin/lectures/blob/master/23/assert.py

THE MOST DIABOLICAL PYTHON ANTIPATTERN

- There are plenty of ways to write bad code. But in Python, one in particular reigns as king

```
try:
    do_something()
except:
    pass
```

⇒ https://realpython.com/the-most-diabolical-python-antipattern/

A COMPLETE EXAMPLE

```
import math

number_list = [10, -5, 1.2, ’apple’]

for number in number_list:
    try:
        number_factorial = math.factorial(number)
    except TypeError:
        print("Factorial is not supported for given input type.")
    except ValueError:
        print("Factorial only accepts positive integer values.",
            number, " is not a positive integer.")
    else:
        print("The factorial of", number, "is", number_factorial)
    finally:
        print("Release any resources in use.")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/23/example.py

VALIDATE USER INPUT

```
def inputNumber(message):
    while True:
        try:
            userInput = int(input(message))
        except ValueError:
            print("Not an integer! Try again.")
            continue
        else:
            return userInput

age = inputNumber("How old are you?")
```

NESTED try

```
try:
    try:
        raise ValueError(’1’)
    except TypeError:
        print("Caught the type error")
except ValueError:
    print("Caught the value error!")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/23/nested_try.py

SUMMING UP

- After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python:
    - `raise` allows you to throw an exception at any time
    - `assert` enables you to verify if a certain condition is met and throw an exception if it isn't
    - In the `try` clause, all statements are executed until an exception is encountered
    - `except` is used to catch and handle the exception(s) that are encountered in the `try` clause
    - `else` lets you code sections that should run only when no exceptions are encountered in the try clause
    - `finally` enables you to execute sections of code that should always run, with or without any previously encountered exceptions

EXERCISES

- Moodle activity at: LE23: Exceptions