Exception handling

Jordi Cortadella and Jordi Petit
Department of Computer Science

Quadratic equation

```
import math

def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    a, b, c = [float(x) for x in open(file).readline().split()]
    discrim = b*b - 4*a*c
    if discrim < 0:
        print('The equation has no real roots!')
    elif discrim == 0:
        root = -b / (2*a)
        print(f'There is a double root at {root}')
    else:
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')


quadratic_equation('coef.txt')
```

Source: John Zelle, *Python programming: an introduction to Computer Science*, Franklin, Beedle & Associates, 2004.

Exception handling

- Very often programs are infested with **if** statements to detect errors and special cases that are produced at runtime.
- It is very difficult to predict all the abnormal situations that can occur during the execution of a program.
- Exception handling is a mechanism that allows to catch errors without messing up the code of the algorithm.
- When an error occurs, and exception is raised, the flow of execution is broken and transferred to an exception handler.
- Examples of exceptions: invalid input data, division by zero, square root of a negative number, overflow, file not found, index out-of-bounds, violation of the pre-condition of a function, etc.

Exceptions in Python

```
try:
    <body>
except <ErrorType>:
    <handler>
```

```
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    try:
        a, b, c = [float(x) for x in open(file).readline().split()]
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')
    except ValueError:
        print('No real roots')
```

Exceptions in Python

What if the exception is not caught?

```
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    a, b, c = [float(x) for x in open(file).readline().split()]
    discRoot = math.sqrt(b*b - 4*a*c)
    root1 = (-b + discRoot) / (2*a)
    root2 = (-b - discRoot) / (2*a)
    print(f'The solutions are: {root1}, {root2}')
```

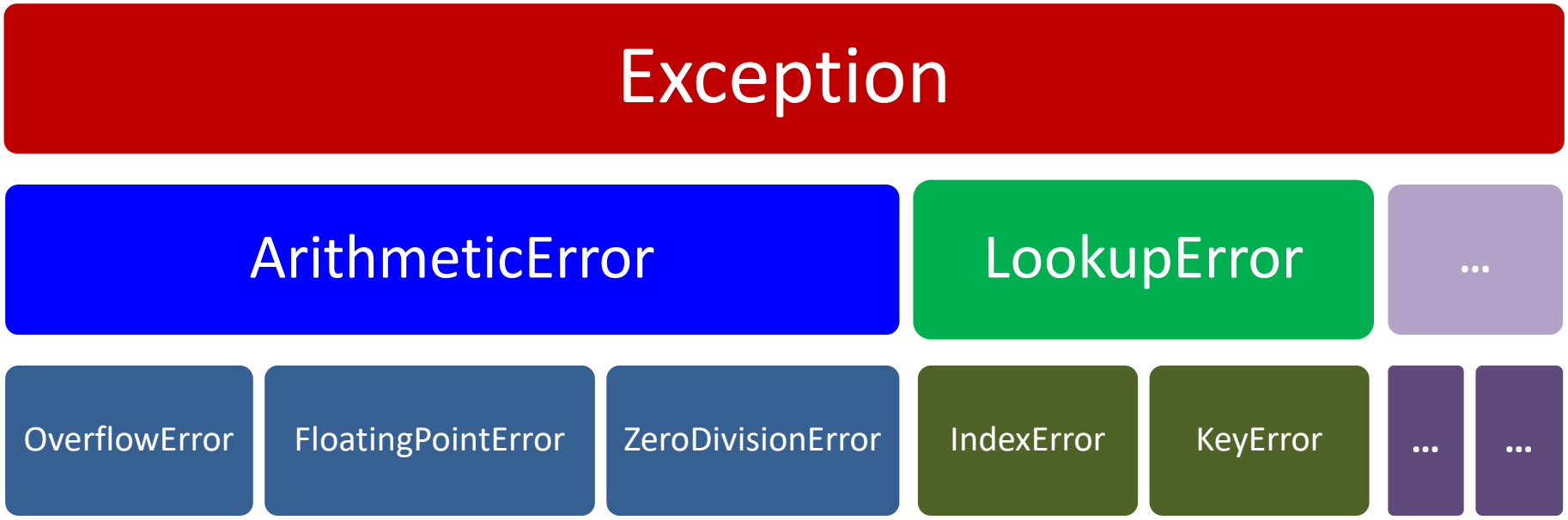
```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "quadratic.py", line 13, in ?
    discRoot = math.sqrt(b*b - 4*a*c)
ValueError: math domain error
```

Unhandled exceptions are finally caught by the Python interpreter

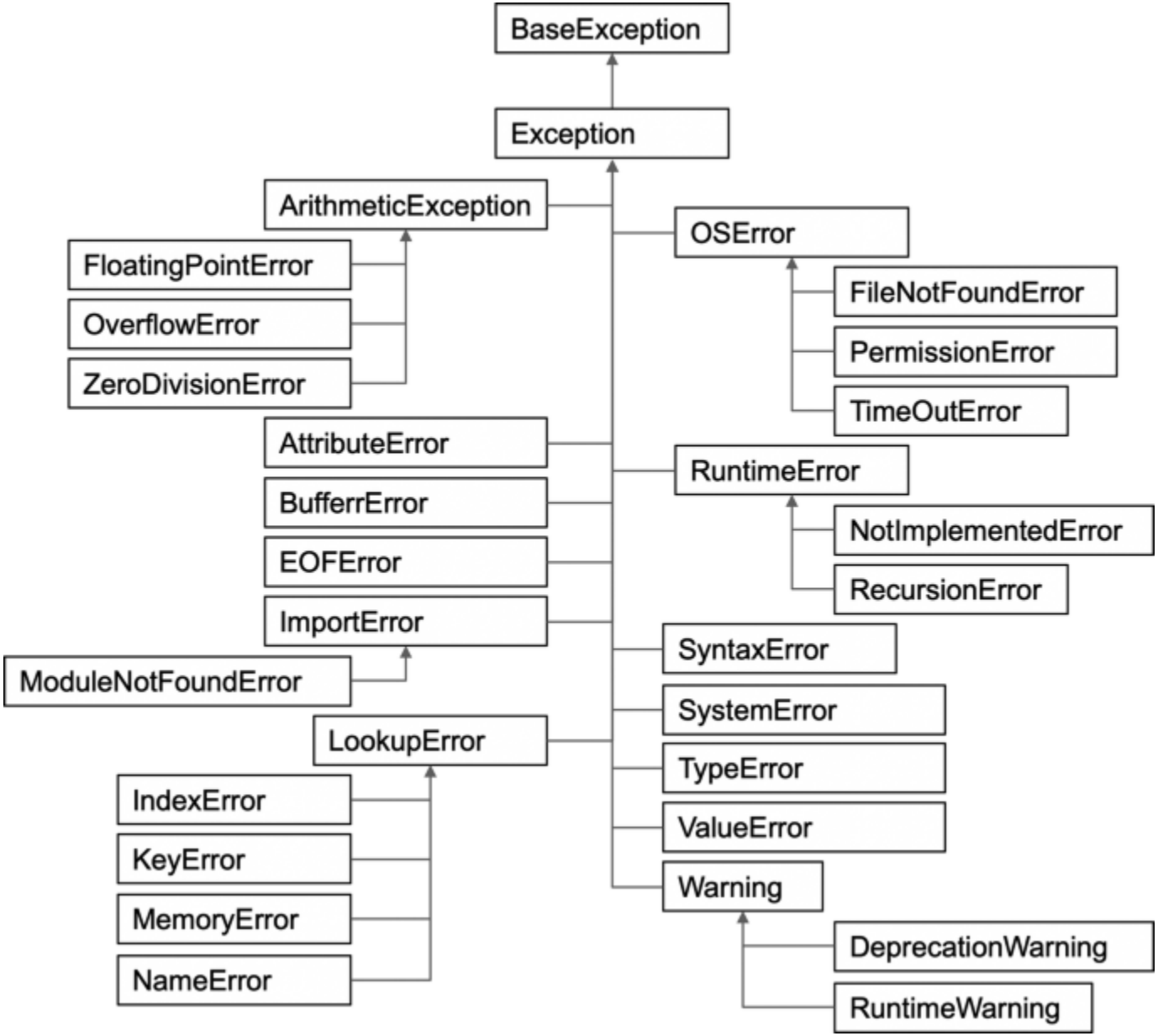
Exception hierarchy

- When an exception is raised, the control is transferred to the innermost **try-except** statement than handles the exception.

- There is an exception hierarchy:



Exception hierarchy



Handling multiple exceptions

```
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    try:
        a, b, c = [float(x) for x in open(file).readline().split()]
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')
    except ValueError as error:
        if str(error) == 'math domain error':
            print('No real roots.')
        else:
            print('Wrong format for coefficients.')
    except FileNotFoundError:
        print(f'File {file} could not be found.')
    except:  # something else, e.g., division by zero
        print('Something went wrong, sorry!')
```

Raising exceptions

```
# The program itself can generate exceptions

if x < 0:
    raise ValueError('No negative numbers allowed')

if not isinstance(s, str):
    raise TypeError('A string is expected')
```

Python allows user-defined exceptions using classes (not explained here), e.g.,

```
class MyError(Exception):
…
```

Assertions

- Assertions are sanity checks used to test the program. They are often used to check the pre-conditions of the functions and detect internal errors of the program, e.g.,

```
def Kelvin2Fahrenheit(temperature: float) -> float:
    assert temperature >= 0, 'Colder than absolute zero!'
    return (Temperature-273)*1.8+32
```

- Assertions are not exceptions, but they raise an **`AssertionError`** that can be caught by an exception handler.

Conclusions

- Use exceptions to maintain clean code and catch abnormal program executions at runtime
- Use assertions for internal sanity-checks
- Print informative error messages:
  - Non-informative message: *"Unexpected error"*
  - Informative message: *"File abc.txt not found"*