PROGRAMMING IN PYTHON I

Exceptions

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: `schoergenhumer@ml.jku.at`

Write mails only for personal questions

Institute ML Homepage

Motivation

- In programming, we sometimes encounter problems that would crash our program
  - Wrong data type used as input by user
  - Use case we did not consider
  - Arithmetic, indexing or other errors in our code
- The severity of such a problem depends on how well the program can handle the error
- Proper error handling can:
  - Give the user clear information on what went wrong
  - Terminate the program in a proper way (e.g., closing all open files, writing a logfile, saving trained ML models, . . .)
  - Fix the error and continue with the program execution (if it makes sense; not always desired!)

Exceptions in Python

- In Python, errors **raise exceptions**
  - □ If an error occurs, an exception is created (“raised”)
  - □ An exception carries information on what went wrong
  - □ There are different exception types (we can also create our own exception types)
- Exceptions can be **caught** and dealt with in the program
- If an exception is raised, the program execution will jump to where the exception is caught or to the end of the program
  - □ In Python, exceptions have a notion of control-flow tools, such as if-else code blocks
  - □ However, don’t overuse exceptions!
- We can raise exceptions ourselves

Exceptions in Python: Syntax

- We can raise an exception with the **raise** statement:
  - This raises a `ValueError` exception:

    ```python
    raise ValueError("Some error message")
    ```

- To catch an exception, we have to be prepared:
  - We have to use a **try** code block, in which we can catch the exception …
  - … followed by an **except** code block, in which we specify our exception handling
  - We can also follow it with a **finally** code block to unconditionally execute code (e.g., for closing/saving files)
- An exception is passed upwards the calling hierarchy (an exception can occur in any (nested) function call) until it is caught somewhere or the program ultimately fails

Predefined Exceptions in Python

- Some common predefined exceptions:
  - □ `TypeError` (incompatible data types)
  - □ `ValueError` (correct type but incorrect value)
  - □ `IndexError` (sequence index out of range)
  - □ `KeyError` (key not in dictionary)
  - □ `ZeroDivisionError`
  - □ `FileNotFoundError`
  - □ `ModuleNotFoundError`
- Many more, full list: `https://docs.python.org/3/library/exceptions.html#bltin-exceptions`

Calling Hierarchy

- Assume we have a function `fun1()` that calls `fun2()` and this function again calls `fun3()`
- Some `ValueError` occurs in `fun3()`
- Now, it is checked if there exists some exception handling for this `ValueError` in the reverse order
  - Check if the exception is caught in `fun3()`
  - If not, jump to the code where `fun3()` was called from `fun2()` and check if the exception is caught there
  - If not, jump to the code where `fun2()` was called from `fun1()` and check if the exception is caught there
  - If not, jump to the code where `fun1()` was called from (e.g., our main script) and check if the exception is caught there
  - If not, the program ends with this exception

Catching with Normal Execution

- Here, we catch an exception, print a warning and continue with our program normally

```python
try:
    a = 1 + "f"  # This will raise a "TypeError"
    a += 2  # This will not be executed
except TypeError as ex:
    # We will land here if "TypeError" was raised
    print(f"We caught the exception {ex}")
    a = 1 + 2
a *= 2  # This will be executed
```

- `as ex` is optional; it allows us to do something with the occurred exception (`ex` is just some identifier)

Catching with Reraising an Exception

- Here, we catch an exception, print a warning and raise the exception again to terminate our program

```
try:
    a = 1 + "f"  # This will raise a "TypeError"
    a += 2  # This will not be executed
except TypeError as ex:
    # We will land here if "TypeError" was raised
    print(f"We caught the exception {ex}")
    # Perform some exception handling code
    raise ex  # Reraise the exception
a *= 2  # This will not be executed
```

- We raised the same exception again, but of course, we could have raised any other (new) exception as well

Output

```
We caught the exception unsupported operand type(s) for +: 'int
    ' and 'str'
Traceback (most recent call last):
  File "C:\Users\andis\example.py", line 8, in <module>
    raise ex  # Reraise the exception
  File "C:\Users\andis\example.py", line 2, in <module>
    a = 1 + "f"  # This will raise a "TypeError"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Contains useful information for debugging:
  - Name of exception
  - Detailed message
  - Traceback (context where the exception occurred)

Catching Multiple Exceptions (1)

- We can catch multiple exceptions as well:

```
try:
    dangerous_fun()
except ValueError as ex:
    # Do something
except TypeError as ex:
    # Do something
except IndexError as ex:
    # Do something
```

- If we want to run the same exception handling code, we can catch all of them at once:

```
try:
    dangerous_fun()
except (ValueError, TypeError, IndexError) as ex:
    # Do something that is common for all the three
    exceptions above
```

Catching Multiple Exceptions (2)

- In case we have multiple `except` clauses, **only one** is ever **executed** (or **none** if the particular exception is not part of any `except` clauses or no exception occurred)
- The evaluation is done from **top to bottom**, the first matching `except` clause is executed
- This means that the order matters for **derived** exceptions[1] (more on this topic when we discuss classes). Example:

```
try:
    1 / 0
except ArithmeticError:
    # Do something
except ZeroDivisionError:
    # Is never executed since "ZeroDivisionError" is
    # a special version of "ArithmeticError", which
    # has already been caught above
```

[1] https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Conditional Code Execution

- In Python, you can also execute code within a `try-except` statement only if no exception occurred by using **else** after the last `except`:

```
try:
    fun()
except ValueError:
    # Do something
else:
    # Only executed if no exception occurred
```

- Useful if you want this conditional execution and better than placing the code within the `try` clause (avoids catching additional exceptions on accident)

Unconditional Code Execution

- If you want some code to be executed independently of whether an an exception occurred or not, you can use **finally** at the end of a `try` statement (`except` clauses are optional in this case)

```
try:
    fun()
except ValueError:
    # Do something
finally:
    # Always executed
```

- Useful if you need to perform some clean-up operations that must always be done (e.g., closing files)
- Note that only the execution is guaranteed, there might still go something wrong (another exception) which causes the `finally` to terminate early without having run all its code

Nested Exception Handling

- Exception handling code can be arbitrarily nested, i.e., you can have further `try` statements in your `except`, `else` and `finally` clauses

```
try:
    fun()
except ValueError:
    try:  # Nested try-except
        ...
    except ...
finally:
    try:  # Nested try-finally
        ...
    finally:
        ...
```

- The same rules apply for all nested exception handling