17 – Exceptions

Bálint Aradi

Scientific Programming in Python (2025)

https://atticlectures.net/scipro/python-2025/

Exceptions

```
mystr = "ab"
val = int(mystr)
```

Where did the error occur?

```
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    val = int(mystr)
ValueError: invalid literal for int() with base 10: 'ab'
```

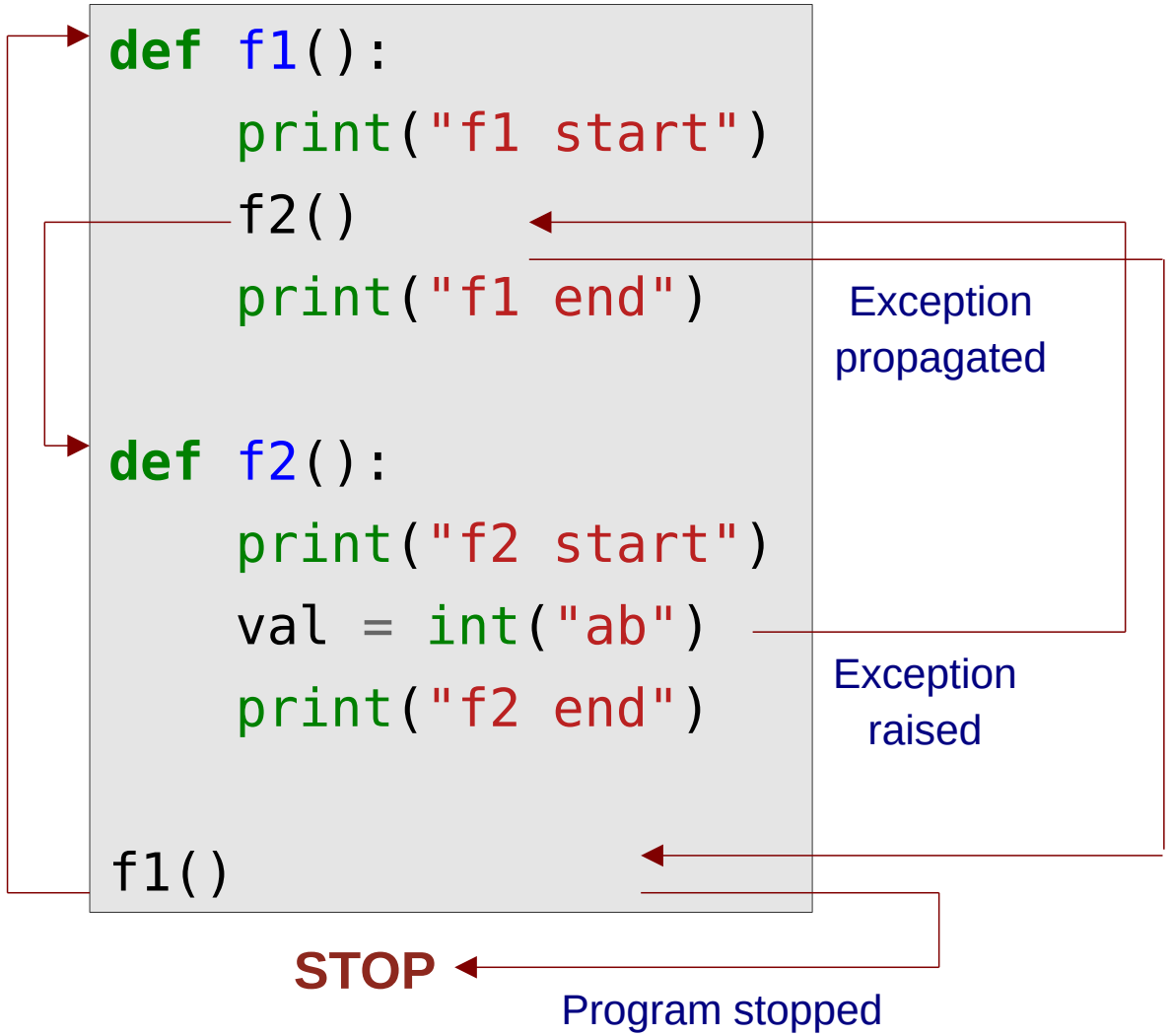
Exception class

Error message

- Exceptions **signalize errors** during code execution
- If an unexpected error happens which Python can not (or does not want to) handle, an **exception is raised**

- Exceptions are part of a **class hierarchy**
- Exception class indicates the kind of error occurred.

Exception propagation




```
f1 start
f2 start
Traceback (most recent call last):
 File "test.py", line 11, in <module>
   f1()
 File "test.py", line 3, in f1
   f2()
 File "test.py", line 8, in f2
   val = int("ab")
         ^^^^^^^^^
ValueError: …
```

- Exception contains **call stack trace** information (how this point of code execution has been reached)
- The **most recent** call is shown **last**

Handling exceptions

```
fname = "missing_file"
with open(fname, "r") as fp:
    txt = fp.read()
```

```
Traceback (most recent call last):
  File "...", line 2, in <module>
    with open(fname, "r") as fp:
FileNotFoundError: [Errno 2] No such file or directory: 'missing_file'
```

- A robust program should handle **exceptions which can be expected**

```
try:
    with open(fname, "r") as fp:
        txt = fp.read()
except FileNotFoundError:
    print(f"Could not open '{fname}'")
    # Recover here or exit
```

```
Could not open 'missing_file'
```

Handling exceptions

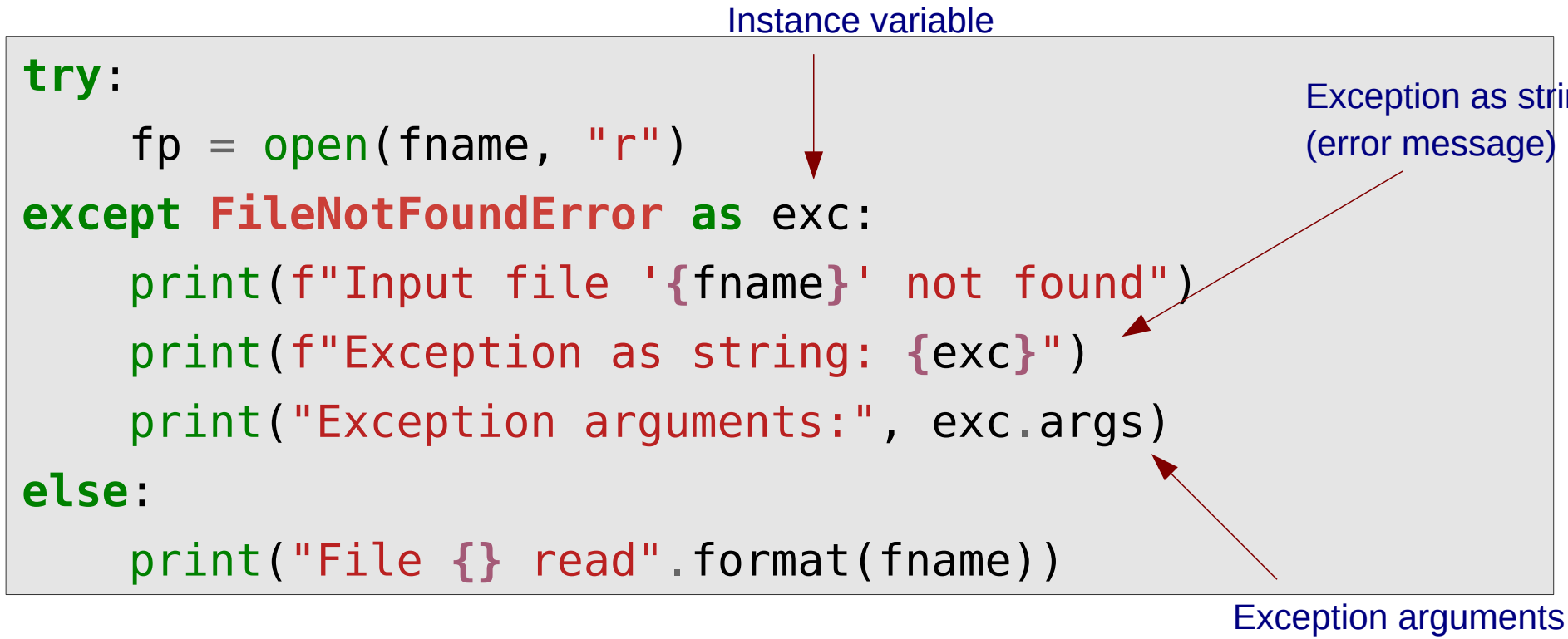
- Exception can be caught and processed with the **try … except …** clause

```
try:
    fp = open(fname, "r")
except FileNotFoundError:
    print(f"Could not open file '{fname}', using default content")
    txt = "default text"
else:
    txt = fp.read()
    fp.close()
    print("File '{fname}' succesfully read".format(fname))
```

- If exception is raised by any statement in the **try** block, it is compared with the exceptions in the **except** clauses
- Block of **first matching exception** will be executed
- If no exception matches, exception is propagated
- The optional **else** block is executed, if no exception occured

Handling exceptions

- The except clause can obtain the exception instance as variable for further inspection




```
try:
    fp = open(fname, "r")
except FileNotFoundError as exc:
    print(f"Input file '{fname}' not found")
    print(f"Exception as string: {exc}")
    print("Exception arguments:", exc.args)
else:
    print("File {} read".format(fname))
```

```
Input file missing_file not found
Exception as string: [Errno 2] No such file or directory: 'missing_file'
Exception arguments: (2, 'No such file or directory')
```

- Number and type of exception arguments are exception dependent

Handling exceptions

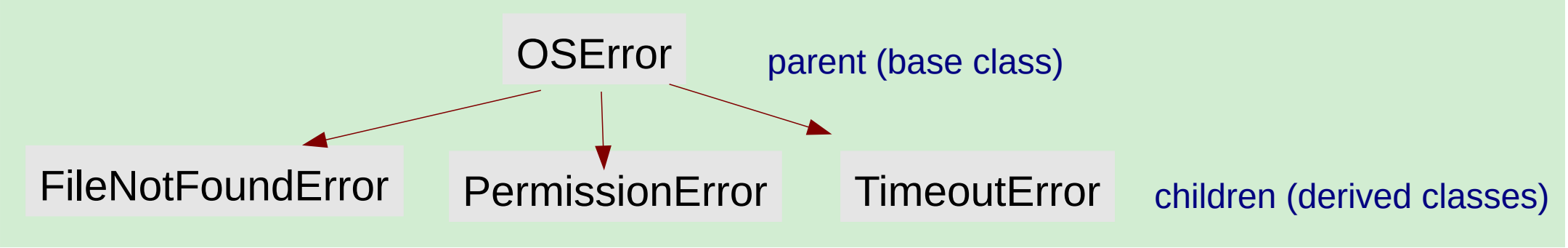
- A **try … except …** construct may contain several except clauses
- If an exception is raised, the **first matching** except clause will be executed

```
try:
    fp = open(fname, "r")
except FileNotFoundError:
    print(f"Input file {fname} not found")
except PermissionError:
    print(f"No read permission for input file {fname}")
```

- There will be maximally one **except** clause executed.

Exception class hierarchy

- Exceptions are organized in a **class hierarchy**
- More specific exceptions (children) inherit from more general exceptions (parents)



- If an exception appears in an **except** clause, it handles the **exception** itself **or any of its descendants** lower in the class hierarchy

```
try:
    fp = open(fname, "r")
except OSError:
    print("Could not open file")
    print("File not present or present but not readable")
```

Exiting gracefully via sys.exit()

- A script can be exited via **sys.exit()**
- The argument of exit is given to the operating system and can be used there to take action depending on the exit code

```
import sys

try:
    with open('input.dat', 'r') as fp:
        content = fp.read()
except OSError:
    print("Could not read input file")
    print("Exiting...")
    sys.exit(1)
```

- **Only the highest level main program/script should call exit**, never lower level functions in a module

Raising an exception

- Your library can signalize irrecoverable errors by raising exceptions
- You have to pass an initialized exception to the raise command
- You can raise Pythons built-in exceptions, if appropriate.
- Most exceptions in Python accept the error message as only argument.

```
if abs(diagelem) < TOL:
    msg = "Singular matrix"
    raise ValueException(msg)
```

- It is also possible to define your own exceptions via inheritance:

```
class LinAlgError(Exception):
    """Signalizes linear algebra problems (e.g. linear dependence)"""
```

User exceptions should be derived from the **Exception** class

Testing exceptions in pytest

- Pytest can test, whether an exception had been raised.
- Code which is supposed to raise an exception must be embedded in the **pytest.raises()** context manager (via **with** construct)

```
def test_passes_if_exception_is_raised():
    with pytest.raises(ValueError):
        gaussian_eliminate(aa_singular, bb)
```

```
def test_passes_if_exception_is_raised():
    with pytest.raises(ValueError, match="Singular matrix"):
        gaussian_eliminate(aa_singular, bb)
```

Checking also the exception message

- The test passes, if the specified exception had been raised during the execution of the context, otherwise it fails.

Be sure to test only for the single **specific exception**, you **expect** to be raised in a given unit test!