Exceptions

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 7A's technical content. See python-exceptions[2] for the full reference on `try`/`except` and raising errors.

Today's outline

- Syntax errors vs. run-time errors (exceptions)
- Common built-in exception types
- `try`/`except`, `else`, `finally`
- Catching specific vs. all exceptions
- Exercise: robust input reading
- Raising exceptions

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-exceptions.pdf

Syntax errors vs. exceptions

When Python *parses* a file, it returns a **syntax error** if the contents don't correspond to valid Python code. Code that passes parsing transitions to **run-time**, where errors (**exceptions**) can occur — these differ from syntax errors because they can't be detected until they trigger.

Run-time errors are bad because they abort execution of the program, and all intermediate work is lost (unless saved to a file).

Common run-time exceptions

Exception	Description
`AssertionError`	an assertion fails
`IOError`	file does not exist
`IndexError`	index out of range
`KeyError`	key in dict does not exist
`NameError`	variable does not exist
`TypeError`	unexpected type is given to a function
`ValueError`	correct type, but inappropriate value
`ZeroDivisionError`	division by zero attempted

```
>>> xs = [1, 2, 3]
>>> xs[4]
IndexError: list index out of range

>>> xs = {'a': 1}
>>> xs['b']
KeyError: 'b'

>>> "two" + 2
TypeError: can only concatenate str (not "int") to str
>>> 2 + "two"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Note that `int(xs: str) -> int`'s type contract means `int("ten")` shouldn't raise a *type* error, because `"ten"` is indeed a string — it's the *value* that's wrong:

```
>>> int("ten")
ValueError: invalid literal for int() with base 10: 'ten'
```

Try/except

```
try:
    <code>   # this code may throw an error
except ExceptionName:
    <code>   # this code is run when the error is of kind ExceptionName
```

(Also called *try-catch* in other languages, e.g. C++.) Typically used in user-facing layers of software and/or when accessing/interacting with external resources (files, databases, API calls, network calls, etc).

```
>>> x = 0
>>> 1/x               # without try-except
ZeroDivisionError: division by zero
```

```
>>> x = 0
>>> try:
...     1/x
... except ZeroDivisionError:
...     print("Please don't divide by zero...")
Please don't divide by zero
```

It doesn't halt execution.

Catching specific vs. all exceptions

We can catch *any* exception without specifying the error type, but this can lead to undiscovered issues — it's bad practice to ignore all errors, so it's better to catch specific error types:

```
>>> x = 0
>>> try:
...     print(y)
...     1/x
... except:
...     print("You did something fishy...")
```

We can also catch among a list of exceptions:

```
>>> try:
...     print(y)     # this throws a name error
...     1/x
... except NameError:
...     print("You're using an undefined name...")
... except ZeroDivisionError:
...     print("You divided by zero...")
```

Exercise: robust input reading

Write a function `def io_double() -> int:` which takes no inputs, but prompts the user for a number and doubles it.

A first attempt:

```
>>> def io_double() -> int:
...     str_x = input("Number please: ")
...     int_x = int(str_x)
...     return 2*int_x

>>> io_double()
Number please: 2
4
>>> io_double()
Number please: two
...
ValueError: invalid literal for int() with base 10: 'two'
```

Wrapping the casting in a `try`/`except` inside a `while True:` loop lets the user retry until they succeed:

```
>>> def io_double() -> int:
...     while True:
...         str_x = input("Number please: ")
...         try:
...             int_x = int(str_x)
...             return 2*int_x   # unreachable if the line above errors
...         except ValueError:
...             print("That wasn't a number! Try again...")

>>> io_double()
```

```
Number please: two
That wasn't a number! Try again...
Number please: 3
6
```

General try/except framework

```
try:
    <code>
except ExceptionName_0:
    <code>
except ExceptionName_1:
    <code>
    ...
except ExceptionName_k:
    <code>
```

Python requires that **specific** exceptions appear before **general** ones:

```
try:
    <code>
except ZeroDivisionError:
    <code>
except:
    <code>
```

`else` and `finally`

```
try:
    <code>       # run the code under try
except:
    <code>       # execute the code under except, when there is an exception
else:
    <code>       # no exceptions? run the code under else, after try
finally:
    <code>       # always run this code
```

```
try:
    x = int(input("Type a number: "))
    y = 1/x
```

```
    # note that if an exception is raised on a line of code here,
    # the following lines don't execute -- keep try blocks as short
    # as possible (don't put everything here!)
except ZeroDivisionError:
    print("Cannot enter zero")
except ValueError:
    print("Must type in a number")
else:
    print(y)
finally:
    print("this block gets always executed")
```

Raising

To *raise* an error (rather than catch it):

```
>>> def safe_div(x: int, y: int) -> float:
...     """ Return 1 / (x-y). """
...     if x == y:
...         raise ZeroDivisionError   # halt execution as soon as zero division
...     return 1/(x-y)

>>> safe_div(1, 1)
ZeroDivisionError


def get_applicant_age() -> int:
    age = int(input("Enter your age: "))
    if age < 15:
        raise ValueError(f"Age must be >= 15 to apply for a license -- currently it's {age}")
    else:
        return age

try:
    get_applicant_age()
except ValueError:
    print("do something - don't issue license, etc")
```

Two exception-handling philosophies

LBYL (Look Before You Leap):

```
if b == 0:
    result = None
    print("zero division!")
else:
    result = a/b
```

EAFP (Easier to Ask for Forgiveness than Permission):

```
try:
    result = a/b
except ZeroDivisionError:
    result = None
    print("zero division!")
```

Summary

We can catch errors that are thrown at run-time with the `try/except` control structure. We should only use a `try/except` when absolutely necessary.

Next: 2025-09-09-scope[3] (Lecture 7B).

[3] courses/csse1001/2025-09-09-scope.pdf