CS303E: Elements of Computers and Programming

Exception Handling

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

© William D. Young, All rights reserved.

Last updated: November 9, 2026 at 09:54

Preventing Runtime Errors

So far we've "handled" errors by either assuming they don't happen, or making sure they don't.

```
# Accept a year from the user.
inputYear  = input("Specify a year: ")
# Let's do some validation on the year.
if inputYear.isdigit() and int(inputYear) > 1752:
    year = int(inputYear)
else:
    print("Bad year entered:", inputYear)
    return
```

If you were careless in your validation, your program might crash!

Python provides an alternative approach: allow the error to happen and *catch* it.

Handling Exceptions

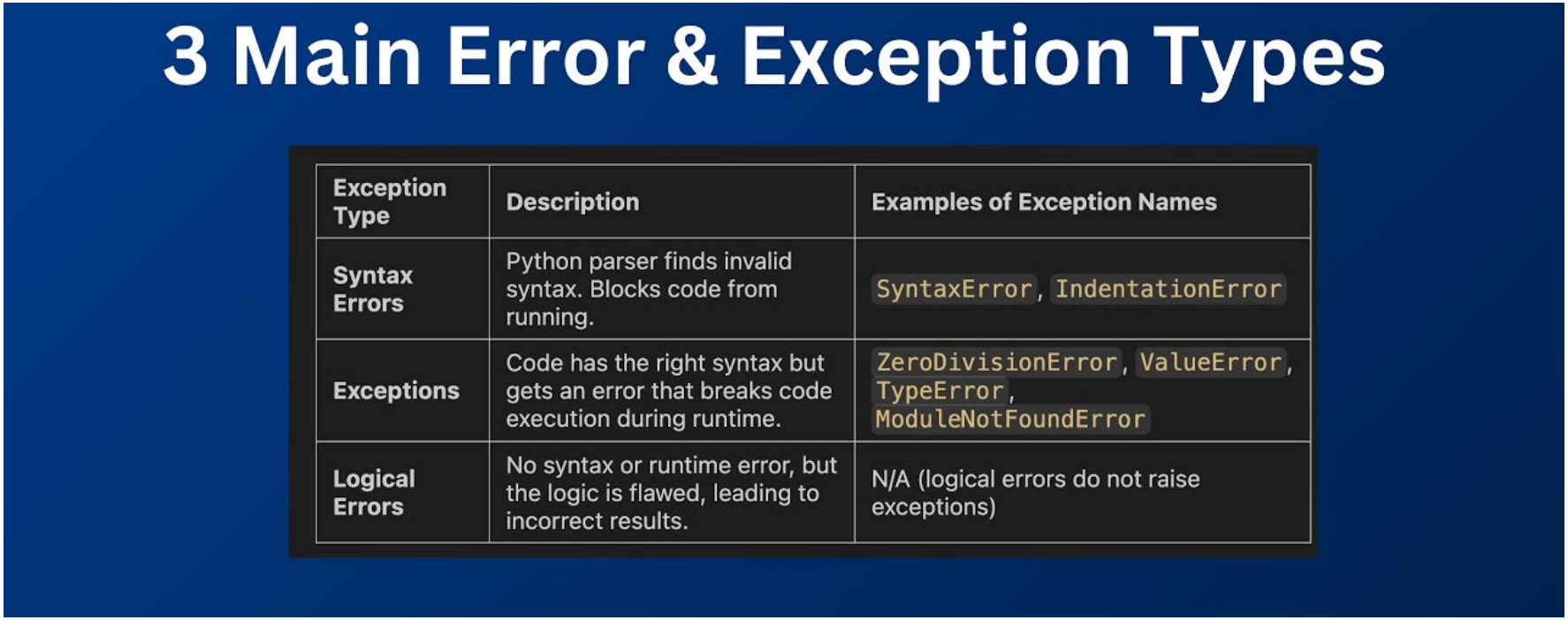


Python exceptions allow handling errors that occur during execution of a program.

Python exceptions only apply for errors in *syntactically correct* code.

The `try-except` construct provides a structured way of recovering from runtime errors without crashing your program.

Errors vs. Exceptions



Programmers often use the words *error* and *exception* interchangeably.

Python distinguishes between Syntax Errors (compile-time/parsing issues) and Exceptions (runtime issues).

Basic Form

A number of runtime exceptions can arise when you're running Python code.

```
>>> x, y = 5, 0
>>> x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> s = "abcd"
>>> s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> v = float( input("Enter a float: ") )
Enter a float: abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'abc'
>>>
```

Handling Exceptions

A `try-except` block lets you *catch* the exception and deal with it, without crashing your program.

```
x, y = 5, 0
try:
    z = x / y
except ZeroDivisionError:
    print("Error: You cannot divide by zero!")
```

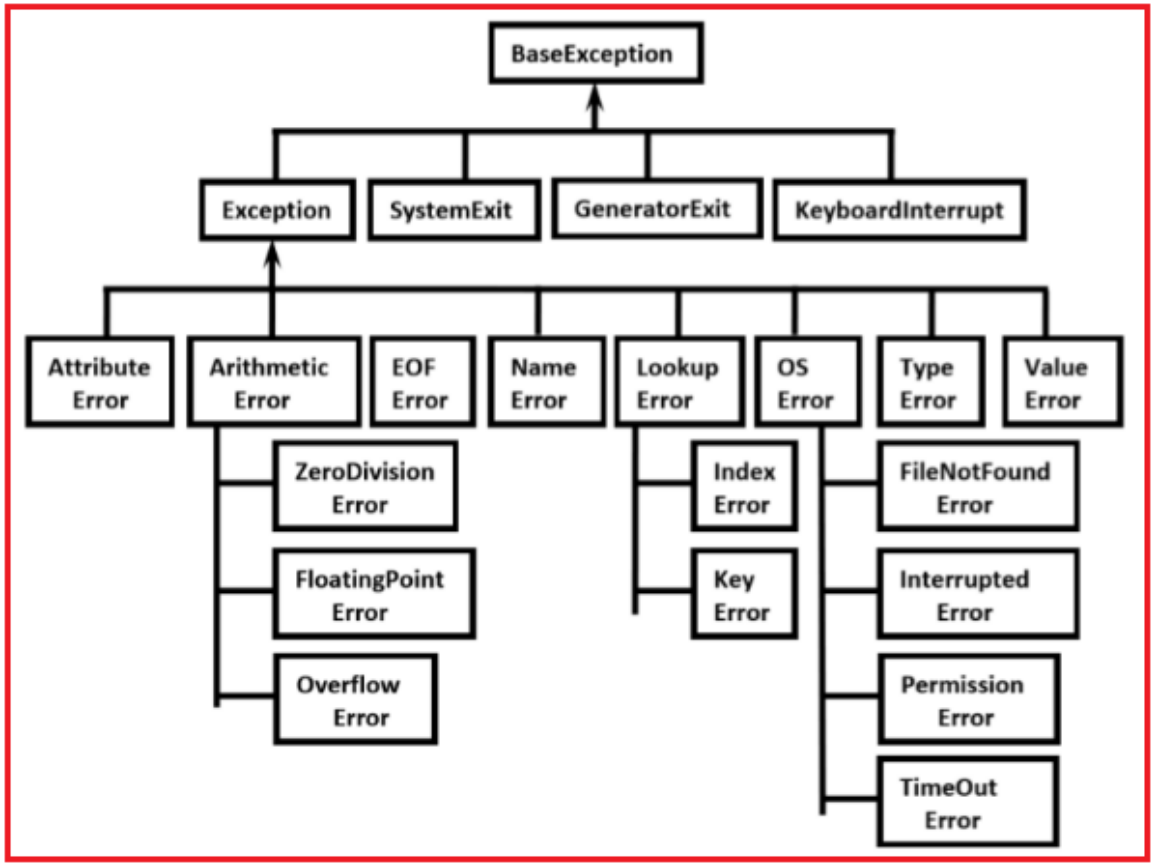
```
> python divideWithError.py
Error: You cannot divide by zero!
>
```

Notice that we "try" to execute the code. If it executes without error we move on to the next statement.

But if the specified error `ZeroDivisionError` occurs, we execute the `except` block.

Python Exceptions

As with everything in Python, exceptions are a class with attributes and methods.



Note that some exceptions are more general than others. E.g., `ZeroDivisionError` is an `ArithmeticError`, but not necessarily vice versa.

Form of a Try-Except

Note that it's possible to have several `except` clauses to catch multiple errors.

General form:

```
try:
   try block statements
except exception-1:
   exception-1 statements
 ...
except exception-n:
    exception-n statements
```

All of the statements must be indented the same amount.

Try-Except Example

```
def divide ():
    """Accept floats n and d from the user and return n / d.
        """
    try:
        n = float(input("Enter a numerator: "))
        d = float(input("Enter a denominator: "))
        result = n / d
        print( result )
    except ZeroDivisionError:
        print("Error: You cannot divide by zero!")
    except ValueError:
        print("Error: Please enter a valid float.")
```

If an exception occurs, except clauses are tried in order; if an exception is not caught it propogates to the surrounding code.

Order Exceptions

A subclass is more specific than its parent class. E.g., `ZeroDivisionError` is a *subclass* of `ArithmeticError` which is a subclass of `Exception`.

```
try:
    ...
except ArithmeticError:
    ...
except ZeroDivisionError:
    ...
```

The second `except` is useless; the error would be caught by the first clause.

Else and Finally Clauses

`try-except` statements can also have `else` and `finally` clauses.

General form:

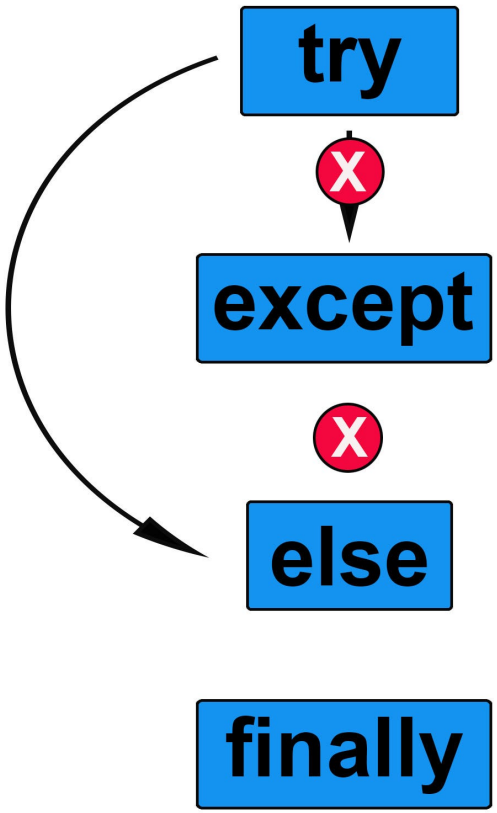
```
try:
   try block statements
except exception-1:
   exception-1 statements
 ...
except exception-n:
    exception-n statements
else:
    else block statements
finally:
    finally block statements
```

The `else` block executes if *no exception* is raised.

The `finally` block always executes. It's great for clean-up tasks like closing files.

Errors and Exceptions





Run the code under try

Execute the code under exception

When there is no exception run else statement after try

Always run the code under finally

try-except-else-finally Example

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: The file does not exist.")
else:
    # Good for code that should run only if
    # the try clause didn't raise an error.
    print("File read successfully!")
    print(content)
finally:
    # This block executes no matter what to ensure
    # the file structure closes safely.
    print("Cleaning up resources...")
```

Beware: exceptions can be raised inside a `try-except` block.

E.g., `file.close()` wouldn't be safe inside the `finally` clause here. It would raise a `NameError` exception if the `FileNotFoundError` occurred.

Bare Except Clause

It is possible to write a "bare" except clause, which catches *every* exception. **Don't do that!**

```
try:
    ...
except:   # Catches EVERYTHING
    print("An error occurred.")
```

1. KeyboardInterrupt (pressing Ctrl-C) and SystemExit are also exceptions. May make it impossible to interrupt or exit your program.
2. Masks real bugs: instead of showing what exception was raised, it hides the issue.

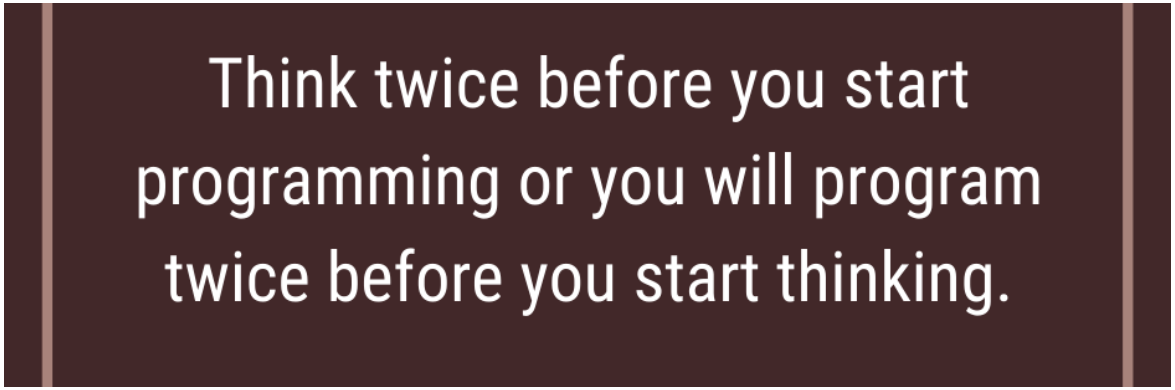
Best Practices



- Keep try blocks small.
- Avoid bare except clauses.
- Order exceptions from specific to general.

This was just a brief tour of `try-except`; there are other features we didn't cover.

Think Before You Code



Don't use `try-except` as a substitute for thinking carefully about your problem.



Next stop: Recursion