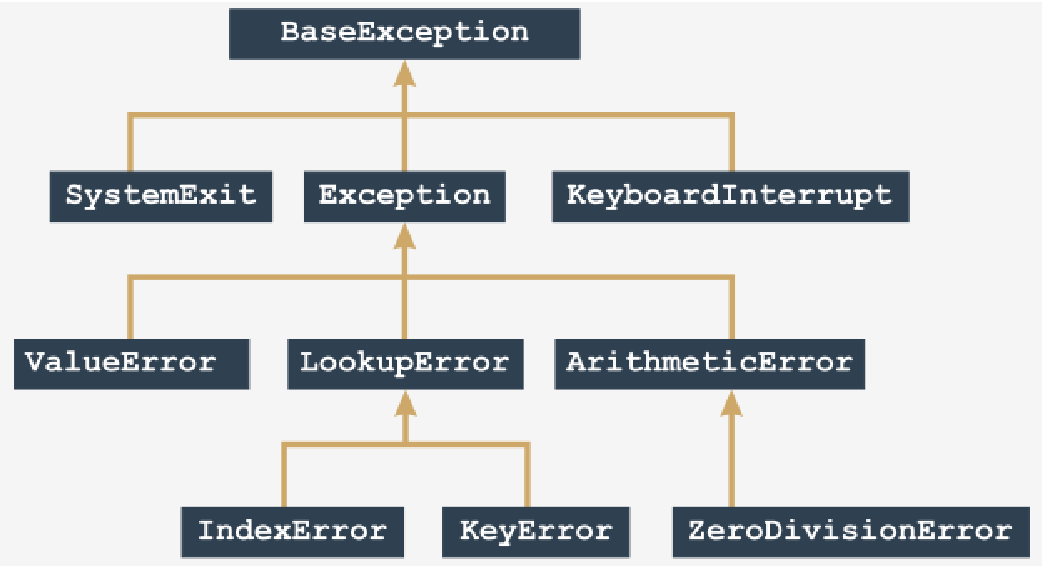
The anatomy of exceptions

Exceptions

Python 3 defines **63 built-in exceptions**, and all of them form a **tree-shaped hierarchy**, although the tree is a bit weird as its root is located on top.

Some of the built-in exceptions are more general (they include other exceptions) while others are completely concrete (they represent themselves only). We can say that **the closer to the root an exception is located, the more general (abstract) it is**. In turn, the exceptions located at the branches' ends (we can call them **leaves**) are concrete.

Take a look at the figure:



It shows a small section of the complete exception tree. Let's begin examining the tree from the `ZeroDivisionError` leaf.

Note:

- `ZeroDivisionError` is a special case of more a general exception class named `ArithmeticError`;
- `ArithmeticError` is a special case of a more general exception class named just `Exception`;
- `Exception` is a special case of a more general class named `BaseException`;

We can describe it in the following way (note the direction of the arrows - they always point to the more general entity):

```
BaseException
↑
Exception
↑
ArithmeticError
↑
ZeroDivisionError
```

We're going to show you how this generalization works. Let's start with some really simple code.

Look at the code below.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Oooppsss...")

print("THE END.")
```

It is a simple example to start with. Run it.

The output we expect to see looks like this:

```
Oooppsss... THE END.
```

Now look at the code below:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Oooppsss...")
print("THE END.")
```

Something has changed in it - we've replaced `ZeroDivisionError` with `ArithmeticError`.

You already know that `ArithmeticError` is a general class including (among others) the `ZeroDivisionError` exception.

Thus, the code's output remains unchanged. Test it.

This also means that replacing the exception's name with either `Exception` or `BaseException` won't change the program's behavior.

Let's summarize:

- each exception raised **falls into the first matching branch**;
- the matching branch doesn't have to specify the same exception exactly - it's enough that the exception is **more general** (more abstract) than the raised one.

Look at the code.

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")
```

What will happen here?

The first matching branch is the one containing `ZeroDivisionError`. It means that the console will show:

```
Zero division!
THE END.
```

Will it change anything if we swap the two `except` branches around? Just like here below:

```
try:
      y = 1 / 0
except ArithmeticError:
      print("Arithmetic problem!")
except ZeroDivisionError:
      print("Zero Division!")

print("THE END.")
```

The change is radical - the code's output is now:

```
Arithmetic problem!
THE END.
```

Why, if the exception raised is the same as previously?
The exception is the same, but the more general exception is now listed first - it will catch all zero divisions too. It also means that there's no chance that any exception hits the `ZeroDivisionError` branch. This branch is now completely unreachable.

Remember:

- the order of the branches matters!
- don't put more general exceptions before more concrete ones;
- this will make the latter one unreachable and useless;
- moreover, it will make your code messy and inconsistent;
- Python won't generate any error messages regarding this issue.

If you want to **handle two or more exceptions** in the same way, you can use the following syntax:

```
try:
      :
except (exc1, exc2):
      :
```

You simply have to put all the engaged exception names into a comma-separated list and not to forget the parentheses.
If an **exception is raised inside a function**, it can be handled:

- inside the function;
- outside the function;

Let's start with the first variant - look at the code below.

```python
def badFun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

badFun(0)

print("THE END.")
```

The `ZeroDivisionError` exception (being a concrete case of the `ArithmeticError` exception class) is raised inside the `badfun()` function, and it doesn't leave the function - the function itself takes care of it.

The program outputs:

```
Arithmetic problem!
THE END.
```

It's also possible to let the exception propagate **outside the function**. Let's test it now.

Look at the code below:

```python
def badFun(n):
    return 1 / n

try:
    badFun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")

print("THE END.")
```

The problem has to be solved by the invoker (or by the invoker's invoker, and so on).

The program outputs:

```
What happened? An exception was raised!
THE END.
```

Note: the **exception raised can cross function and module boundaries**, and travel through the invocation chain looking for a matching `except` clause able to handle it.

If there is no such clause, the exception remains unhandled, and Python solves the problem in its standard way - **by terminating your code and emitting a diagnostic message**.

Now we're going to suspend this discussion, as we want to introduce you to a brand new Python instruction.

The `raise` instruction raises the specified exception named `exc` as if it was raised in a normal (natural) way:

`raise exc`

Note: `raise` is a keyword.

The instruction enables you to:

- **simulate raising actual exceptions** (e.g., to test your handling strategy)
- partially **handle an exception** and make another part of the code responsible for completing the handling (separation of concerns).

Look at the code below.

```
def badFun(n):
    raise ZeroDivisionError

try:
    badFun(0)
except ArithmeticError:
    print("What happened? An error?")

print("THE END.")
```

This is how you can use it in practice.

The program's output remains unchanged.

In this way, you can **test your exception handling routine** without forcing the code to do stupid things.

The `raise` instruction may also be utilized in the following way (note the absence of the exception's name):

`raise`

There is one serious restriction: this kind of `raise` instruction may be used **inside the `except` branch** only; using it in any other context causes an error.

The instruction will immediately re-raise the same exception as currently handled.

Thanks to this, you can distribute the exception handling among different parts of the code.

Look at the code below.

```python
def badFun(n):
    try:
        return n / 0
    except:
        print("I did it again!")
        raise

try:
    badFun(0)
except ArithmeticError:
    print("I see!")

print("THE END.")
```

Run it - we'll see it in action.

The `ZeroDivisionError` is raised twice:

- first, inside the `try` part of the code (this is caused by actual zero division)
- second, inside the `except` part by the `raise` instruction.

In effect, the code outputs:

```
I did it again!
```

```
I see!
```

```
THE END.
```

Now is a good moment to show you another Python instruction, named `assert`. This is a keyword.

```
assert expression
```

How does it work?

- It evaluates the expression;
- if the expression evaluates to `True`, or a non-zero numerical value, or a non-empty string, or any other value different than `None`, it won't do anything else;
- otherwise, it automatically and immediately raises an exception named `AssertionError` (in this case, we say that the assertion has failed)

How it can be used?

- you may want to put it into your code where you want to be **absolutely safe from evidently wrong data**, and where you aren't absolutely sure that the data has been carefully examined before (e.g., inside a function used by someone else)
- raising an `AssertionError` exception secures your code from producing invalid results, and clearly shows the nature of the failure;
- **assertions don't supersede exceptions or validate the data** - they are their supplements.

If exceptions and data validation are like careful driving, assertion can play the role of an airbag.

Let's see the `assert` instruction in action. Look at the code below.

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

Run it.

The program runs flawlessly if you enter a valid numerical value greater than or equal to zero; otherwise, it stops and emits the following message:

```
Traceback (most recent call last):

    File ".main.py", line 4, in

        assert x >= 0.0

AssertionError
```