Python Essentials 2:
Module 5
Modules, packages string and list methods, and exceptions

In this module, you will learn about:

Python modules: their rationale, function, how to import them in different ways, and present the content of some standard modules provided by Python;
the way in which modules are coupled together to make packages.
the concept of an exception and Python's implementation of it, including the try-except instruction, with its applications, and the raise instruction.
strings and their specific methods, together with their similarities and differences compared to lists.

(part 10)

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:

```
BaseException
    SystemExit
    Exception
        ValueError
        LookupError
            IndexError
            KeyError
        ArithmeticError
            ZeroDivisionError
    KeyboardInterrupt
```

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.

Exceptions: continued

Look at the code in the editor. It is a simple example to start with. Run it.

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

print("THE END.")
```

The output we expect to see looks like this:

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

output

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.

Exceptions: continued

Look at the code in the editor. What will happen here?

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

print("THE END.")
```

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

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

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.
```
output

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.

Exceptions: continued

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 in the editor.

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

bad_fun(0)

print("THE END.")
```

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

The program outputs:

Arithmetic problem!
THE END.
output

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

Look at the code below:

```
def bad_fun(n):
    return 1 / n

try:
    bad_fun(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.
output

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.

Exceptions: continued

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 in the editor. This is how you can use it in practice.

```
def bad_fun(n):
    raise ZeroDivisionError


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

print("THE END.")
```

The program's output remains unchanged.

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

Exceptions: continued

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 in the editor. Run it - we'll see it in action.

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


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

print("THE END.")
```

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.
```

Exceptions: continued

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 in the editor. Run it.

```
import math

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

x = math.sqrt(x)

print(x)
```

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
```

Key takeaways

1. You cannot add more than one anonymous (unnamed) except branch after the named ones.

```
:
# The code that always runs smoothly.
:
try:
    :
    # Risky code.
    :
except Except_1:
    # Crisis management takes place here.
except Except_2:
    # We save the world here.
except:
    # All other issues fall here.
:
# Back to normal.
:
```

2. All the predefined Python exceptions form a hierarchy, i.e. some of them are more general (the one named BaseException is the most general one) while others are more or less concrete (e.g. IndexError is more concrete than LookupError).

You shouldn't put more concrete exceptions before the more general ones inside the same except branche sequence. For example, you can do this:

```
try:
    # Risky code.
except IndexError:
    # Taking care of mistreated lists
except LookupError:
    # Dealing with other erroneous lookups
```

but don't do that (unless you're absolutely sure that you want some part of your code to be useless)

```
try:
    # Risky code.
except LookupError:
    # Dealing with erroneous lookups
except IndexError:
    # You'll never get here
```

3. The Python statement raise ExceptionName can raise an exception on demand. The same statement, but lacking ExceptionName, can be used inside the try branch only, and raises the same exception which is currently being handled.

4. The Python statement assert expression evaluates the expression and raises the AssertError exception when the expression is equal to zero, an empty string, or None. You can use it to protect some critical parts of your code from devastating data.

Exercise 1

What is the expected output of the following code?

```
try:
    print(1/0)
except ZeroDivisionError:
    print("zero")
except ArithmeticError:
    print("arith")
except:
    print("some")
```

Exercise 2

What is the expected output of the following code?

```
try:
    print(1/0)
except ArithmeticError:
    print("arith")
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```

Exercise 3

What is the expected output of the following code?

```
def foo(x):
    assert x
    return 1/x


try:
    print(foo(0))
except ZeroDivisionError:
    print("zero")
except:
    print("some")
```