# Errors – the programmer

# Errors, failures, and other plagues

**Anything that can go wrong, will go wrong**.

This is Murphy's law, and it works everywhere and always. Your code's execution can go wrong, too. If it can, it will.

Look the code below.

```
1  import math
2
3  x = float(input("Enter x: "))
4  y = math.sqrt(x)
5
6  print("The square root of", x, "equals to", y)
```

There are at least two possible ways it can "go wrong". Can you see them?

- As a user is able to enter a completely arbitrary string of characters, **there is no guarantee that the string can be converted into a float value** - this is the first vulnerability of the code;
- the second is that the `sqrt()` **function fails if it gets a negative argument**.

You may get one of the following error messages.

Something like this:

```
Enter x: Abracadabra
Traceback (most recent call last):
      File "sqrt.py", line 3, in
            x = float(input("Enter x: "))
ValueError: could not convert string to float: 'Abracadabra'
```

Or something like this:

```
Enter x: -1
Traceback (most recent call last):
      File "sqrt.py", line 4, in
            y = math.sqrt(x)
ValueError: math domain error
```

Can you protect yourself from such surprises? Of course you can. Moreover, you have to do it in order to be considered a good programmer.

# Exceptions

Each time your code tries to do something wrong/foolish/irresponsible/crazy/unenforceable, Python does two things:

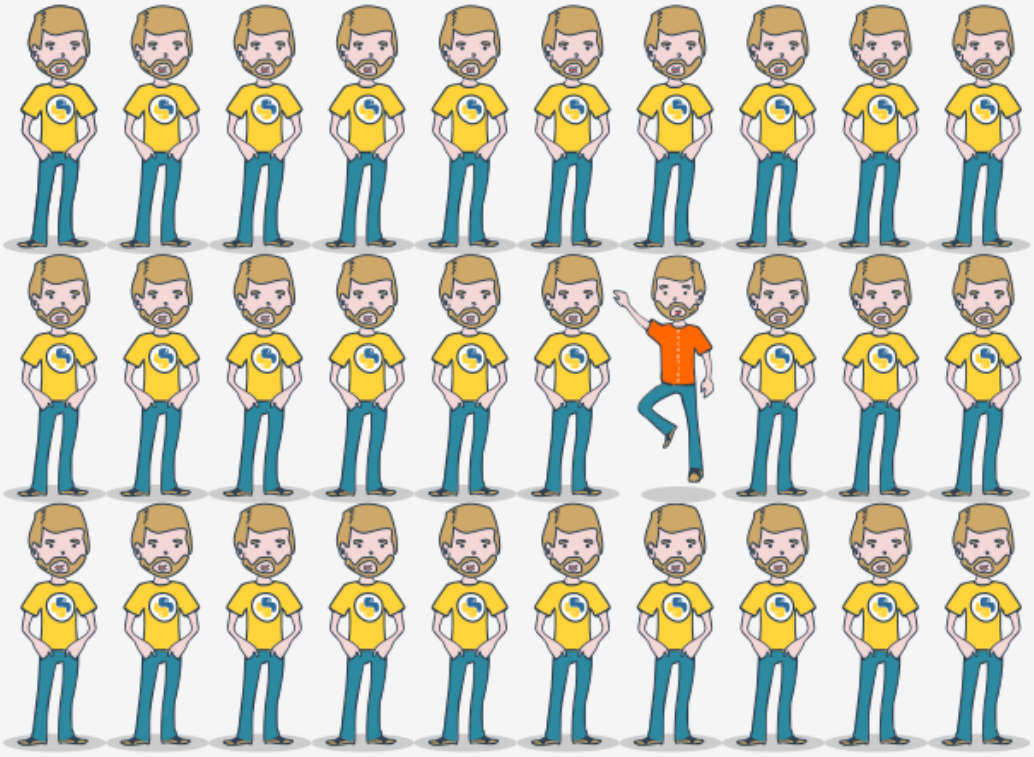
- it **stops your program**;
- it creates a special kind of data, called an **exception**.

Both of these activities are called **raising an exception**. We can say that Python always raises an exception (or that an **exception has been raised**) when it has no idea what do to with your code.

What happens next?

- the raised exception expects somebody or something to notice it and take care of it;
- if nothing happens to take care of the raised exception, the program will be **forcibly terminated**, and you will see an **error message** sent to the console by Python;
- otherwise, if the exception is taken care of and **handled** properly, the suspended program can be resumed and its execution can continue.

Python provides effective tools that allow you to **observe exceptions, identify them and handle them** efficiently. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately.



You know some exception names already.

Take a look at the following diagnostic message:

```
ValueError: math domain error
```

The word in red is just the **exception name**. Let's get familiar with some other exceptions.

Look at the code below.

```
1  value = 1
2  value /= 0
```

Run the (obviously incorrect) program.

You will see the following message in reply:

```
Traceback (most recent call last):
File "div.py", line 2, in
value /= 0
ZeroDivisionError: division by zero
```

This exception error is called `ZeroDivisionError`.

Look at the code below.

```
1  list = []
2  x = list[0]
```

What will happen when you run it? Check.

You will see the following message in reply:

```
Traceback (most recent call last):
File "lst.py", line 2, in
x = list[0]
IndexError: list index out of range
```

This is the `IndexError`.

How do you **handle** exceptions? The word `try` is key to the solution.

What's more, it's a keyword, too.

The recipe for success is as follows:

- first, you have to ***try* to do something**;
- next, you have to **check whether everything went well**.

But wouldn't it be better to check all circumstances first and then do something only if it's safe?

Just like this example.

```
firstNumber = int(input("Enter the first number: "))
secondNumber = int(input("Enter the second number: "))

if secondNumber != 0:
    print(firstNumber / secondNumber)
else:
    print("This operation cannot be done.")

print("THE END.")
```

Admittedly, this way may seem to be the most natural and understandable, but in reality, this method doesn't make programming any easier. All these checks can make **your code bloated and illegible**.

Python prefers a completely different approach.

Look at the code below.

```
firstNumber = int(input("Enter the first number: "))
secondNumber = int(input("Enter the second number: "))

try:
    print(firstNumber / secondNumber)
except:
    print("This operation cannot be done.")

print("THE END.")
```

This is the favorite Python approach.

Note:

- the `try` keyword **begins a block of the code** which may or may not be performing correctly;
- next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution;
- the `except` keyword starts a piece of code which will be **executed if anything inside the `try` block goes wrong** - if an exception is raised inside a previous `try` block, **it will fail here**, so the code located after the except keyword should provide an **adequate reaction** to the raised exception;
- returning to the previous nesting level ends the **try-except** section.

Run the code and test its behavior.

Let's summarize this:

```
try:
    :
    :
except:
    :
    :
```

- in the first step, Python tries to perform all instructions placed between the `try:` and `except:` statements;
- if nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the `except:` block, and the block's execution is considered complete;
- if anything goes wrong inside the `try:` and `except:` block, the execution immediately jumps out of the block and into the first instruction located after the `except:` keyword; this means that some of the instructions from the block may be silently omitted.

Look at the code below.

```
try:
    print("1")
    x = 1 / 0
    print("2")
except:
    print("Oh dear, something went wrong...")

print("3")
```

It will help you understand this mechanism.

This is the output it produces:

```
1
Oh dear, something went wrong...
3
```

Note: the `print("2")` instruction was lost in the process.

This approach has one important disadvantage - if there is a possibility that more than one exception may skip into an `except:` branch, you may have **trouble figuring out what actually happened**.

Just like in our code below.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

Run it and see what happens.

The message: `Oh dear, something went wrong...` appearing in the console says nothing about the reason, while there are two possible causes of the exception:

- non-integer data entered by the user;
- an integer value equal to `0` assigned to the `x` variable.

Technically, there are two ways to solve the issue:

- build two consecutive `try-except` blocks, one for each possible exception reason (easy, but will cause unfavorable code growth)
- use a more advanced variant of the instruction.

It looks like this:

```
try:
            :
except exc1:
            :
except exc2:
            :
except:
            :
```

This is how it works:

- if the `try` branch raises the `exc1` exception, it will be handled by the `except exc1:` block;
- similarly, if the `try` branch raises the `exc2` exception, it will be handled by the `except exc2:` block;
- if the `try` branch raises any other exception, it will be handled by the unnamed `except` block.

Let's move on to the next part of the course and see it in action.

Look at the code below.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print("You cannot divide by zero, sorry.")
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

Our solution is there.

The code, when run, produces one of the following four variants of output:

- if you enter a valid, non-zero integer value (e.g., `5`) it says:

  ```
  0.2
  THE END.
  ```

- if you enter `0`, it says:

  ```
  You cannot divide by zero, sorry.
  THE END.
  ```
- if you enter any non-integer string, you see:

  ```
  You must enter an integer value.
  THE END.
  ```
- (locally on your machine) if you press Ctrl-C while the program is waiting for the user's input (which causes an exception named `KeyboardInterrupt`), the program says:

  ```
  Oh dear, something went wrong...
  THE END.
  ```

Don't forget that:

- the `except` branches are searched in the same order in which they appear in the code;
- you must not use more than one except branch with a certain exception name;
- the number of different `except` branches is arbitrary - the only condition is that if you use `try`, you must put at least one `except` (named or not) after it;
- the `except` keyword must not be used without a preceding `try`;
- if any of the `except` branches is executed, no other branches will be visited;
- if none of the specified `except` branches matches the raised exception, the exception remains unhandled (we'll discuss it soon)
- if an unnamed `except` branch exists (one without an exception name), it has to be specified as the last.

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Let's continue the experiments now.
Look at the code below.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

We've modified the previous program - we've removed the `ZeroDivisionError` branch.

What happens now if the user enters `0` as an input?

As there are **no dedicated branches** for division by zero, the raised exception falls into the **general (unnamed) branch**; this means that in this case, the program will say:

```
Oh dear, something went wrong...
THE END.
```

Try it yourself. Run the program.

Let's spoil the code once again.

Look at the program below.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")

print("THE END.")
```

This time, we've removed the unnamed branch.

The user enters `0` once again and:

- the exception raised won't be handled by `ValueError` - it has nothing to do with it;
- as there's no other branch, you should to see this message:

```
Traceback (most recent call last):
File "exc.py", line 3, in
y = 1 / x
ZeroDivisionError: division by zero
```

You've learned a lot about exception handling in Python. In the next section, we will focus on Python built-in exceptions and their hierarchies.