An exception is a signal that a condition has occurred that can’t be easily handled using the normal flow-of-control of a Python program. Exceptions are often defined as being “errors” but this is not always the case. All errors in Python are dealt with using exceptions, but not all exceptions are errors.

All errros are exceptions viceversa not true.

------------------------------

To explain what an exception does, let’s review the normal “flow of control” in a Python program. In normal operation Python executes statements sequentially, one after the other. For three constructs, if-statements, loops and function invocations, this sequential execution is interrupted.

* For if-statements, only one of several statement blocks is executed and then flow-of-control jumps to the first statement after the if-statement.
* For loops, when the end of the loop is reached, flow-of-control jumps back to the start of the loop and a test is used to determine if the loop needs to execute again. If the loop is finished, flow-of-control jumps to the first statement after the loop.
* For function invocations, flow-of-control jumps to the first statement in the called function, the function is executed, and the flow-of-control jumps back to the next statement after the function call.

Do you see the pattern? If the flow-of-control is not purely sequential, it always executes the first statement immediately following the altered flow-of-control. That is why we can say that Python flow-of-control is sequential. But there are cases where this sequential flow-of-control does not work well.

**Exceptions provide us with way way to have a non-sequential point where we can handle something out of the ordinary (exceptional).**

**----------------------**

The try/except control structure provides a way to process a run-time error and continue on with program execution. Until now, any run-time error, such asking for the 8th item in a list with only 3 items, or dividing by 0, has caused the program execution to stop. In the browser ActiveCode windows, you get an error message in a box below. When you are executing python programs from the command-line, you also get an error message saying something about what went wrong and what line it occurred on. After the run-time error is encountered, the python interpreter does not try to execute the rest of the code. You have to make some change in your code and rerun the whole program.

**With try/except, you tell the python interpreter:**

**Try to execute a block of code, the “try” clause.**

* **If the whole block of code executes without any run-time errors, just carry on with the rest of the program after the try/except statement.**
* **If a run-time error does occur during execution of the block of code:**

**skip the rest of that block of code (but don’t exit the whole program)**

**execute a block of code in the “except” clause**

**then carry on with the rest of the program after the try/except statement**

**----------------------------**

The syntax is fairly straightforward. The only tricky part is that after the word except, there can optionally be a specification of the kinds of errors that will be handled. The catchall is the class Exception**. If you write except Exception: all runtime errors will be handled. If you specify a more restricted class of errors, only those errors will be handled;** any other kind of error will still cause the program to stop running and an error message to be printed.

**If we catch only IndexEror, and we actually have a divide by zero error, the program does stop executing.**

\*\*\*\*\*\*\*IMP CODE\*\*\*\*\*\*\*\*

try:

items = ['a', 'b']

third = items[2]

print("This won't print")

except IndexError as e:

print("error 1")

print(e)

print("continuing")

try:

x = 5

y = x/0

print("This won't print, either")

except IndexError as e:

print("error 2")

print(e)

print("continuing again")

-------------------------------------

When to use ?

The reason to use try/except is when you have a code block to execute that will sometimes run correctly and sometimes not, depending on conditions you can’t foresee at the time you’re writing the code.

For example, when you are running code that fetches data from a website, you may run the code when you don’t have a network connection or when the external website is temporarily not responding. If your program can still do something useful in those situations, you would like to handle the exception and have the rest of your code execute.

As another example, suppose you have fetched some nested data from a website into a dictionary d. When you try to extract specific elements, some may be missing: d may not include a particular key, for example. If you anticipate a particular key potentially not being present, you could write an if..else check to take care of it.

---------------------------------------

It’s considered poor practice to catch all exceptions this way. Instead, python provides a mechanism to specify just certain kinds of exceptions that you’ll catch (for example, just catching exceptions of type KeyError, which happens when a key is missing from a dictionary.