19.1. What is an exception?

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.

19.2. Exception Handling Flow-of-control

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).

19.2.1. Raising and Catching Errors

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

**try**:

<**try** clause code block>

**except** <ErrorType>:

<exception handler code block>

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.

The code below causes an error of type IndexError, by trying to access the third element of a two-element list.

items = ['a', 'b']

third = items[2]

​

The code below causes an error of type ZeroDivisionError, or less specifically ArithmeticError.

x = 5

y = x/0

​

Let’s see what happens if we wrap some of this problematic code in a try/except statement. Note that this won't print doesn’t print: when the error is encountered, the rest of the try block is skipped and the exception block is executed. When the except block is done, it continues on with the nex line of code that’s outdented to the same level as the try: continuing is printed.

try:

items = ['a', 'b']

third = items[2]

print("This won't print")

except Exception:

print("got an error")

​

print("continuing")

​

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

try:

items = ['a', 'b']

third = items[2]

print("This won't print")

except IndexError:

print("error 1")

​

print("continuing")

​

try:

x = 5

y = x/0

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

except IndexError:

print("error 2")

​

​

print("continuing again")

​

There’s one other useful feature. The exception code can access a variable that contains information about exactly what the error was. Thus, for example, in the except clause you could print out the information that would normally be printed as an error message but continue on with execution of the rest of the program. To do that, you specify a variable name after the exception class that’s being handled. The exception clause code can refer to that variable name.

try:

items = ['a', 'b']

third = items[2]

print("This won't print")

except Exception, e:

print("got an error")

print(e)

print("continuing")

​

**Check your understanding**

exceptions-1: Which type of error can be noticed and handled using try/except?

Top of Form

A. syntax  
B. run-time  
C. semantic

Bottom of Form

exceptions-2: When a run-time exception of type ZeroDivisionError occurs, and you have a statement except IndexError, the program will stop executing completely.

Top of Form

A. True  
B. False

Bottom of Form

exceptions-3: After a run-time exception is handled by an except clause, the rest of the code in the try clause will be executed.

Top of Form

A. True  
B. False

Bottom of Form

exceptions-4: How many lines will print out when the following code is executed?

**try**:

**for** i **in** range(5):

**print**(1.0 / (3-i))

**except** Exception, error\_inst:

**print**("Got an error", error\_inst)

Top of Form

A. 0  
B. 1  
C. 3  
D. 4  
E. 5

# 19.3. 👩‍💻 When to use try/except

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.

**if** somekey **in** d:

*# it's there; extract the data*

extract\_data(d)

**else**:

skip\_this\_one(d)

However, if you’re extracting lots of different data, it can get tedious to check for all of them. You can wrap all the data extraction in a try/except.

**try**:

extract\_data(d)

**except**:

skip\_this\_one(d)

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.

**try**:

extract\_data(d)

**except** KeyError:

skip\_this\_one(d)

# 19.4. Standard Exceptions

Most of the standard exceptions built into Python are listed below. They are organized into related groups based on the types of issues they deal with.

| **Language Exceptions** | **Description** |
| --- | --- |
| StandardError | Base class for all built-in exceptions except StopIteration and SystemExit. |
| ImportError | Raised when an import statement fails. |
| SyntaxError | Raised when there is an error in Python syntax. |
| IndentationError | Raised when indentation is not specified properly. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| UnboundLocalError | Raised when trying to access a local variable in a function or method but no value has been assigned to it. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| LookupError | Base class for all lookup errors. |
| IndexError | Raised when an index is not found in a sequence. |
| KeyError | Raised when the specified key is not found in the dictionary. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| MemoryError | Raised when a operation runs out of memory. |
| RecursionError | Raised when the maximum recursion depth has been exceeded. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |

| **Math Exceptions** | **Description** |
| --- | --- |
| ArithmeticError | Base class for all errors that occur for numeric calculation. You know a math error occurred, but you don’t know the specific error. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| FloatingPointError | Raised when a floating point calculation fails. |
| ZeroDivisonError | Raised when division or modulo by zero takes place for all numeric types. |

| **I/O Exceptions** | **Description** |
| --- | --- |
| FileNotFoundError | Raised when a file or directory is requested but doesn’t exist. |
| IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Also raised for operating system-related errors. |
| PermissionError | Raised when trying to run an operation without the adequate access rights. |
| EOFError | Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |

| **Other Exceptions** | **Description** |
| --- | --- |
| Exception | Base class for all exceptions. This catches most exception messages. |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| AssertionError | Raised in case of failure of the Assert statement. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, it causes the interpreter to exit. |
| OSError | Raises for operating system related errors. |
| EnvironmentError | Base class for all exceptions that occur outside the Python environment. |
| AttributeError | Raised in case of failure of an attribute reference or assignment. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

All exceptions are objects. The classes that define the objects are organized in a hierarchy, which is shown below. This is important because the parent class of a set of related exceptions will catch all exception messages for itself and its child exceptions. For example, an ArithmeticError exception will catch itself and all FloatingPointError, OverflowError, and ZeroDivisionError exceptions.

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StopAsyncIteration

+-- ArithmeticError

| +-- FloatingPointError

| +-- OverflowError

| +-- ZeroDivisionError

+-- AssertionError

+-- AttributeError

+-- BufferError

+-- EOFError

+-- ImportError

+-- LookupError

| +-- IndexError

| +-- KeyError

+-- MemoryError

+-- NameError

| +-- UnboundLocalError

+-- OSError

| +-- BlockingIOError

| +-- ChildProcessError

| +-- ConnectionError

| | +-- BrokenPipeError

| | +-- ConnectionAbortedError

| | +-- ConnectionRefusedError

| | +-- ConnectionResetError

| +-- FileExistsError

| +-- FileNotFoundError

| +-- InterruptedError

| +-- IsADirectoryError

| +-- NotADirectoryError

| +-- PermissionError

| +-- ProcessLookupError

| +-- TimeoutError

+-- ReferenceError

+-- RuntimeError

| +-- NotImplementedError

| +-- RecursionError

+-- SyntaxError

| +-- IndentationError

| +-- TabError

+-- SystemError

+-- TypeError

+-- ValueError

| +-- UnicodeError

| +-- UnicodeDecodeError

| +-- UnicodeEncodeError

| +-- UnicodeTranslateError

+-- Warning

+-- DeprecationWarning

+-- PendingDeprecationWarning

+-- RuntimeWarning

+-- SyntaxWarning

+-- UserWarning

+-- FutureWarning

+-- ImportWarning

+-- UnicodeWarning

+-- BytesWarning

+-- ResourceWarning

Bottom of Form