

Errors, Exceptions and try/except:

The following is an edited version of Sections 8.1 through 8.4 of the Python Manual at:

<https://docs.python.org/3.5/tutorial/errors.html>

You have seen lots of error messages in the course so far. There are (at least) two distinguishable kinds of errors: **syntax errors** and **exceptions**.

8.1. Syntax Errors

Syntax errors, also known as **parsing errors** or **compile-time errors**, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
      File "<stdin>", line 1, in ?
          while True print('Hello world')
                          ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function `print()`, since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

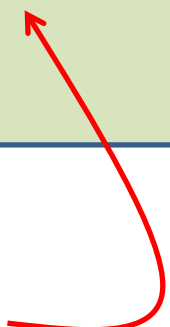
8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. **Errors detected during execution are called exceptions** and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages printed on the Console. For example, when the following program runs:

```
def main():
    example1_IndexError()

def example1_IndexError():
    my_list = [16, 209, 33]
    my_list[3]

main()
```



It breaks on the line:

```
my_list[3]
```

because the list has only 3 items in it, so trying to access the item at index 3 is past the end of the list. This causes the following to get printed on the Console:

Traceback (most recent call last):

File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 11, in <module>

main()

File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 3, in main

example1_IndexError()

File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 9, in example1_IndexError

my_list[3]

IndexError: list index out of range

The **last line of the error message indicates what happened**, that is, what event occurred that the program could not handle. When such an event occurs, the information about what happened is stored in an object of type **Exception**.

Exceptions come in different types, and the type is printed as the first part of the message: the type in this example is **IndexError** (circled in black in the example) – a particular kind of **Exception**. There are lots of different built-in Exception types, including **ZeroDivisionError**, **NameError** and **TypeError**. A complete list of them is in Section 5.2 of:

<https://docs.python.org/3/library/exceptions.html>

Visit that page now and skim section 5.2 in it to get a sense of the various built-in Exception types. You will see only built-in Exceptions in this course; user-defined Exceptions are a follow-up topic to which we won't get.

The words that follow the type of the Exception are words that describe the Exception in more detail. Here are some examples of the words that some Exception types use:

IndexError: list index out of range

ZeroDivisionError: division by zero

NameError: name 'spam' is not defined

TypeError: Can't convert 'int' object to str implicitly

The words for each Exception type can use the specifics of what caused the Exception. For example, the **TypeError** exception is saying that on the line that broke, there is an object of type **int** in that line to which the code tried to apply **str** implicitly. It takes practice to decipher such details; you will get such practice during a forthcoming in-class project.

The preceding part of the error message shows the context where the exception happened, in the form of a **stack traceback**: a list of the function/method calls that led to the line at which the Exception occurred. In the example we are using (*repeated to the right for your convenience, this time with line numbers for the code*):

1. The program started on line 9 of the program:

main()

The traceback shows that line and provides a link (in blue) to that line.

Always use the blue links to get to the lines in a stack traceback; then you can't go wrong.

2. At line 2 of the program, *main* called the next function that led to the exception:

example1_IndexError()

3. The exception occurred inside that function at line 9 of the program:

my_list[3]

In general, the **stack traceback** shows the *sequence of function calls* that lead directly to the line of code that generated the Exception.

```
1 def main():
2     example1_IndexError()
3
4
5 def example1_IndexError():
6     my_list = [16, 209, 33]
7     my_list[3]
8
9 main()
```

Traceback (most recent call last):

[File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 9, in <module>](#)

main()

[File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 2, in main](#)

example1_IndexError()

[File "C:\EclipseWorkspaces\csse120-development\Session19 Input Files Exceptions\src\m0 examples of exceptions .py", line 9, in example1_IndexError](#)

my_list[3]

IndexError: list index out of range

8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions, using **try .. except** statements (called *try/catch* in Java and some other languages). Here is the basic idea, with details to follow:

```
try:
    <stuff that normally works fine>
    <stuff that normally works fine>
    ...
except Exception:
    <whatever you want to happen
       when something in try breaks>

<more stuff>
<more stuff>
...
```

The code in the **try** block executes.

1. If all happens normally, when execution reaches the end of the **try** block, the **except** block is skipped and execution continues below the **try/except** (at *more-stuff* in the above).
2. If ANY statement in the **try** block causes an Exception to occur, execution jumps to the **except** block and does whatever is there. Execution then continues with the *more-stuff*.

There are many reasons why programmers handle some of the possible exceptions in their code. For Python, one reason is that Python philosophically prefers **EAFP** (“Easier to Ask for Forgiveness than Permission”) to **LBYL** (“Look Before You Leap”). So instead of a bunch of IF statements that check that the stuff WILL work fine (LBYL), the code just **TRIES** the stuff, and if it breaks, it deals with it then in the **except** block (EAFP).

Here is an example in which the **except** block prints an error message to the user, then asks the user to try again:

```
while True:
    try:
        x = float(input('Enter a number: '))
        break
    except ValueError:
        print('Oops! That was NOT a number!')
        print('Try again...')

# <do stuff here with x (the number) >
```

If the user enters a number (as hoped-for and expected), the call to the **float** function works fine, then the program encounters the **break** statement, then it leaves the **while** loop and continues on its merry way. If the user enters something other than a number (e.g. **'five'**), execution jumps to the **except** class, which prints a message to the user. Execution then continues inside the loop, thus giving the user the opportunity to try again to enter a number.

In the previous example, the programmer chose to handle the `ValueError` that results when the *float* function is given a non-number. In this next example, the programmer chose to handle any error that results if the program tries to open and read from a file that it can't (perhaps because the file does not exist).

```
try:
    filename = 'some_file.txt'
    f = open(filename, 'r')
    text = f.read()
    # <do stuff with the text in the file>
    f.close()

except OSError:
    print('Could not open & read the file:')
    print(filename)
    print('Check whether you are')
    print('in the right folder!')
    raise
```

If the file is opened successfully, execution *skips* the *except* block and continues normally. If the file could not be opened, execution *jumps* to the *except* block and:

- prints a message that the programmer thinks would be helpful to whomever is running this program, and
- “raises” the Exception again, thus “passing” the same Exception to the calling code (more on this shortly).

The above example does not deal with the file-handling in the best way but it illustrates the general idea of printing (or logging) some information “on the way” to the code that actually handles the Exception. Which brings us to ...

... how Exceptions really work. **When an Exception occurs:**

1. The interpreter (which is running the program) looks in the *last-executed function* (i.e., the one that was executing when the program broke) for a *try/except* clause that **encloses the statement that broke**. If it finds one, execution continues in the *except* clause of that *try/except*.
2. If the interpreter does *not* find an enclosing *try/except* in the last-executed function, the interpreter backs up to *the line of code that called* the last-executed function, in the *second-to-last-executed* function. The interpreter looks in *that* function for a *try/except* clause that encloses *that* line. If it finds one, execution continues in the *except* clause of *that try/except*.
3. If the interpreter does *not* find an enclosing *try/except* in the *second-to-last-executed* function, it proceeds to the *third-to-last-executed function* and proceeds similarly. And so forth.
4. If this process continues all the way back to the line that called *main* and no *try/except* is found along the way, **the interpreter prints a message per the Exception on the Console**. That is what you have been seeing in YOUR programs!