

CSC 241 Notes
Yosef Mendelsohn

With many thanks to Dr. Amber Settle for the original version of these notes.

Prof Settle's notes in turn are based on the Perkovic text.

## Exceptions

There are two basic types of errors that can occur when running a Python program.

**Syntax errors** are errors that are due to the incorrect format of a Python statement.

Syntax errors occur while the statement or program is being parsed (read) but before the code actually gets executed.

A Python Virtual Machine tool called a "parser" discovers these errors.

>>> (3+4]

SyntaxError: invalid syntax

>>> if x == 5

SyntaxError: invalid syntax

>>> print 'hello'

SyntaxError: invalid syntax

>>> lst = [4;5;6]

SyntaxError: invalid syntax

**Exceptions** occur during the execution of the statement or program.

Exceptions do not occur because of a malformed Python statement or program but rather because the program execution gets into an erroneous state.

For example, the following exception is generated when our program attempts to divde by 0:

>>> x = 4

>>> y = 0

>>> **x/y**

Traceback (most recent call last):

 File "<pyshell#7>", line 4, in <module>

 4/0

**ZeroDivisionError**: int division or modulo by zero

The following is an exception generated by an **invalid list index**:

>>> lst = [14, 15, 16]

>>> lst[3]

Traceback (most recent call last):

 File "<pyshell#9>", line 1, in <module>

 lst[3]

**IndexError**: list index out of range

This is an exception generated by an **unassigned variable name**:

>>> x+5

Traceback (most recent call last):

 File "<pyshell#10>", line 1, in <module>

 x+5

**NameError**: name 'x' is not defined

The following is an exception generated by an **illegal value**:

>>> int('4.5')

Traceback (most recent call last):

 File "<pyshell#11>", line 1, in <module>

 int('4.5')

**ValueError**: invalid literal for int() with base 10: '4.5'

In each case an error occurs because the statement execution got into an invalid state. When this happens, we say that the Python interpreter "**raises an exception**".

When an exception is raised, a special 'exception object' gets created. This object will contain all the information relevant to the error.

For example, it will contain the error message that indicates what happened and the program (module) line number at which the error occurred.

When an error occurs, Python's default behavior is for the program to “crash”. At this point, a message containing all of the exception object's information is typically printed to the console.

Every exception object has a specific type that is related to the type of error. Examples include those demonstrated above: ZeroDivisionError, IndexError, NameError, TypeError, ValueError. There are many other types of exceptions out there.

### Catching and handling exceptions

At this stage in your programming careers, if you are working on code your program crashes, no damage is done. You simply go back to your code and work on fixing the error.

But imagine if you are writing code that runs the anesthesia machines at a hospital. If an exception occurs, the *last* thing you want is for the program to crash and the machine to stop functioning!

Or suppose the software crashed for banking ATMs, international trading, the International Space Station, the Mars Rover right before landing, etc, etc. All of these could have devastating ramifications.

Even a small company can fail if their software crashes resulting in the compromise or loss of user data.

**Fortunately, the whole point of the exception paradigm is that it allows the programmer the opportunity to "handle" the exception and to try and resolve it in such a way that prevents the program from crashing.**

When an exception object gets created, the following takes place:

1. The normal execution flow of the program stops
2. The execution switches to the so-called "exceptional control flow"

The *default* exceptional control flow is to stop the program and print the error message contained in the exception object.This isn't much different than a "crash" since your program will stop working.

**However, we as programmers can (and should!) change this default exception behavior.**  We do this using a set of clauses called "**try** and **except**":

Look at the following code:

>>> strAge = input('Enter your age: ')

Enter your age: 22

>>> age = int(strAge)

The above example will be fine – as long as the user types something that Python can successfully convert to an int.

But what if it can't? Look at what happens if the user provides something the program can't handle:

>>> strAge = input('Enter your age: ')

Enter your age: fifteen

**>>> age = int(strAge)**

**Traceback (most recent call last):**

 **File "<pyshell#15>", line 1, in <module>**

 **age = int(strAge)**

**ValueError: invalid literal for int() with base 10: 'fifteen'**

Fortunately, we can use 'exception handling' to deal with exceptions that arise –without allowing our program to crash.

Here is a far better version of the above code that handles the exception that was generated:

strAge = input('Enter your age: ')

**try:**

 age = int(strAge)

**except:**

 print('Enter your age using digits 0-9!')

Let's run this code:

Enter your age: fifteen

**Enter your age using digits 0-9!**

At least our program didn't crash this time! Very soon we will learn how to do things like loop our code until the user gives us a valid input for age.

### Multiple except clauses

Exceptions come in all kinds of types and variations.

Here is an even better version of our earlier example In this version we test for a specific type of exception:

>>> strAge = input('Enter your age: ')

Enter your age: fifteen

try:

 age = int(strAge)

except **ValueError:**

 print('Enter your age using digits 0-9!')

Enter your age using digits 0-9!

Recall the comment above that refers to the idea that the type of error that is generated should match the name listed after one of the exception statements.

The point is that **a try statement may have more than one except clause**.

This is useful as it allows us to specify different handlers depending on the type of exception that is generated.

Also, if you want, you can have a single except statement name multiple exceptions as a parenthesized tuple:

except (RuntimeError, TypeError, NameError):

 #code to handle these exceptions...

When an exception occurs, only one handler is allowed to be executed. Once one handler is executed, any remaining exception blocks are skipped.

If you have a bunch of named exception clauses, you might want to leave the **final** excepetion clause un-named. This allows you to have code that will kick in if an exception of some type that you didn't anticipate is generated. Remember that if an exception is generated, the program crashes – unless you find a way to "handle" it. So having this "wild-card" exception can be useful.

**Even if you can't write code to save your program, you can at least use this last exception clause to print an error message and, perhaps, re-raise the exception.** We will discuss this idea of re-rasing an exception at a later point.

try:

f = open('mfile2.txt', 'r')

s = f.readline()

i = eval(s)

except (IOError, FileNotFoundError):

print('There was an error accessing the file.')

except ValueError:

print('Could not convert data to a number.')

except:

print('An unexpected error has occurred.')

**Summary of how exception handling works:**

When an exception is encountered while executing the body of a try statement, the Python interpreter will jump to the body of the except statement. We call this block of code the exception "handler".

A try/exception block works as follows:

* First, the **try clause** is executed.
	+ Note: The try clause refers to the statements(s) inside the try block.
* If no exception occurs, the except clause is skipped and the execution of the try statement is finished.
* If an exception occurs during execution of the try block, the remainder of the try block is skipped. At this point, if the type of error that was generated matches the exception named after the except keyword, the except block is executed and then execution continues after the try statement.
* If an exception occurs which does *not* match the exception named in the except clause, it is passed on to outer try statements. (We will discuss what is meant by 'outer' try statements later). If no handler is found, it is an **unhandled exception** and execution stops with a generic message.