# PALs CSC-152

Section 02, Professor Kim, M-W-F

Tony Stone

November 12, 2025

# Lesson VII

## 1 Motivation

No, not even close. While we have now comfortable rooted ourselves in the processes of reading and writing to files, you will come to appreciate the almost enigmatic world of programming, and the sheer immensity of all that can go wrong.

But have no fear! We have many tools at our disposal that will enable us to combat. These concepts are traditionally taught in tandem. Often file handling serves as a means to explore **exception handling**, or the process of cleanly dealing with the errors that arise when executing code. I find that this approach often leads to them being tied together in an almost synonymous relationship. But we already have motivation to work with files as we discussed in the last lesson! Our motivation to learn about file handling exceeds far past the application of file handling. Thus, the topics are provided in separate lessons. Some of the examples features here will involve files, but also other kinds of issues that may come up!

### 1.1 Breaking the Fourth Wall

If you will allow me a moment to speak to you directly and candidly; more often than not you will not be using some exception handling. The code that you plug into a python terminal to quickly test a code snippet will not require exception handling. A majority of assignments you will encounter in coursework, or future educational endeavors will not require exception handling. Exception handling is meant to be a way to make your code water-tight.

Suppose you wanted a bullet-proof vehicle, and you started by steel-plating a go-kart frame. How nonsensical! The go-kart must be built up into something like a vehicle first; something reliable with a few features. After that, we can begin thinking about steel plating, otherwise it is overkill. 90% of the programming you will do is riding on go-karts. Save Exception handling for when you plan to juice one up.

Also, also, I am referring to 'Exception' as the stuff that goes wrong in code. There is a difference between an Error, and Exception, a this and a that. But the Juice of diving into it is not worth the squeeze. When we say **Exception**, we are talking about the stuff that goes wrong.

## 2 Exception Handling

### 2.1 the Try Statement

The canonical math 'no-no' is division by zero. Let's do it!

```
1 print(3/0)
```

Output:

```
    print(3/0)
          ~^~
ZeroDivisionError: division by zero
```

Unsurprisingly, this is a no-can do. Now is a good time to introduce the concept of a **stack trace**. For whatever reason, a stack trace is not typically acknowledged until software engineering-type courses, but I find that extremely unhelpful. Your stack Trace is the tool that lets you parse whatever caused your program to crash. Please spend some time just simply reading it. Even if you do not pull a single thing from it, get familiar with the format, and try to follow what it is reporting.

Anyway, the simple division method simply will not do. It never will, but we don't need it too, we just need our program to keep chugging. The **try** statement is what will let us do that. The **try block** is the section of sketchy code we put after a try statement. Its almost like asking our program to perform a task, instead of forcing it.

However a try statement is not a standalone structure, it needs to have a corresponding keyword that helps the program stay on it's feet. We have some options, keywords **finally**, and **except**. Finally is used to mark a section of code that will execute regardless of the execution of the try block. This can be as easy as a pass statement, but we'll print a message for clarity.

```
print("\nExample Two:")
try:
    print(3/0)
finally:
    print("Program executed")
```

Output:

```
Program executed
...
    print(3/0)
          ~^~
ZeroDivisionError: division by zero
```

As you can see, our program executes, but it still delivers us all of the stack trace jargon. This is the purpose of the except keyword. By OOP convention, we 'catch' the error, and we can format it such that it is more useful to us. When an exception is **raised**, we'll take it and give it a name, and format it into a message.

```
try:
        print(3/0)
    except Exception as e:
        print(e,"happened")
```

```
    finally:
        print("Program executed")
```

Output:

```
division by zero happened
Program executed
```

Here, we call our exception 'e', and with our print statement, we are informed of our erroneous division in a much more readable way. Also, a standalone except statement is totally fine, the finally statement just allows you some more control.

## 2.2 The Dark Arts

I emphasized 'raised' but completely blew past it. In general, 'raise' is simply the term we use for an error that comes out of executing code. However, for any aspiring mage seeking to dabble in the dark arts, raise is in fact a keyword. You can practice tried-and-true arcana (Python Documentation) to raise an Error stock to Python, or you can take to the tomes and artifice new ones (write a class).

Let's look into raising our own exceptions. The ones stock to Python are almost certainly more than enough, we'll leave exception crafting to the thaumaturges.

We'll write a function, but we'll be picky, and raise an error if we deem the input unsuitable. Consider the case of a function that adds two numbers. If our user inputs a string (which remember, they will) we need to handle that.

```
def addNum(a, b):
    if type(a) is str or type(b) is str:
        raise ValueError("That is not a number!")
    return a + b

sum = None
try:
    sum = addNum(1, 2)
except ValueError:
    print("Ugh-oh")
finally:
    print(sum)
```

Output:

```
3
```

Now for the case in which our user inputs a string, regardless of the safeguards and warnings put in place beforehand.

```
def addNum(a, b):
    if type(a) is str or type(b) is str:
        raise ValueError("That is not a number!")
    return a + b

sum = None
try:
    sum = addNum(1, "astring")
except Exception:
    print("Ugh-oh")
finally:
    print(sum)
```

Output:

```
Ugh-oh
None
```

Please put a more helpful message than "ugh-oh," but this clearly indicates that a problem has arisen. We then show that sum has not updated with the finally block. Note, the error we raise, and the try-except/finally logic is separate. We also raise a **ValueError** specifically, just make sure that the error you raise, and the error you handle is cohesive to some extent. We'll skip the definitions, but ValueError is a subclass of Exception, so we can simply except Exception, and ValueError will neatly fit into that.

# 3 Back to File Handling

Let's take what we've learned about exception in general handling and relate it back to the much more useful application of file handling. Let's (not) open the file 'nonExistantFile.txt.'

```
file = open('basicFolder/nonExistantFile.txt', 'r')
contents = file.read()
print(contents)
file.close()
```

Output:

```
    file = open('basicFolder/nonExistantFile.txt', 'r')
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory:
'basicFolder/nonExistantFile.txt'
```

Imagine trying to download a wealth of data for a project, and right in the middle of it this happens. Well we don't have to, because we have acquired the tools we need to navigate such an event.

```
fileName = 'basicFolder/nonExistantFile.txt'
contents = None
try:
    file = open(fileName, 'r')
    contents = file.read()
    file.close()
except FileNotFoundError:
    print(fileName, "doesn't exist")
finally:
    print(contents)
```

Output:

```
basicFolder/nonExistantFile.txt doesn't exist
None
'basicFolder/nonExistantFile.txt'
```

Just for our sanity, let's look back and make sure this could work, with our old 'substantFile.txt.'

```
fileName = 'basicFolder/substantFile.txt'
contents = None
try:
    file = open(fileName, 'r')
    contents = file.read()
    file.close()
except FileNotFoundError:
    print(fileName, "doesn't exist")
finally:
    print(contents)
```

Output:

```
1 2 3 4 5
Howdy
:D
```

# 4 Key Terms

- **Exception Handling**: The process of dealing with exceptions.
- **Exception**: A problem that is prone to crashing code.
- **Stack Trace**: Report of the errors in the code, specifically a traced path to the original error.
- **try**: The keyword to organize exception prone code.

- **except**: The keyword which actually handles the exception, and allows for an alternative course of action in the case of an exception.
- **finally**: The keyword which executes a block of code regardless of the outcome from the try-except block.
- **raise**: The word for the realization of an exception, (a keyword if done intentionally).

# 5 Availability

- In Lecture:
  - Wednesday: 10:00am - 11:00am
- PAL Sessions:
  - Monday: 2:00pm - 3:00pm
  - Wednesday: 9:00am - 10:00am
  - Wednesday: 2:00pm - 3:00pm

# References


[1] Tony Gaddis. *Starting Out with Python*. Pearson, 5th edition, 2021.

[2] GeeksforGeeks. Python Tutorial — Learn Python Programming Language - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/python/python-programming-language-tutorial/. [Accessed 29-10-2025].

[3] Bradley N. Miller and David L. Ranum. Problem solving with algorithms and data structures using python. https://runestone.academy/ns/books/published/pythonds/index.html, 2014. Interactive online edition — accessed: 2024.

[4] Python Software Foundation. *Python: A dynamic, open source programming language*, 2025. Version 3.x documentation.

[5] W3Schools. W3Schools.com — w3schools.com. https://www.w3schools.com/python/default.asp. [Accessed 29-10-2025].