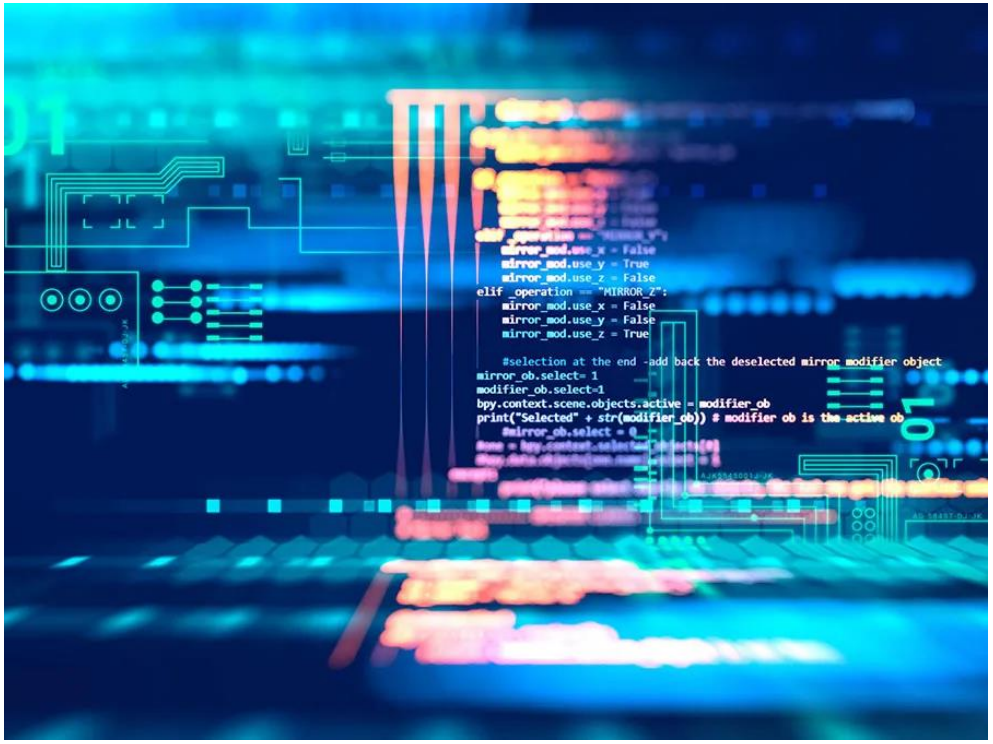
Lecture 12: Exceptions

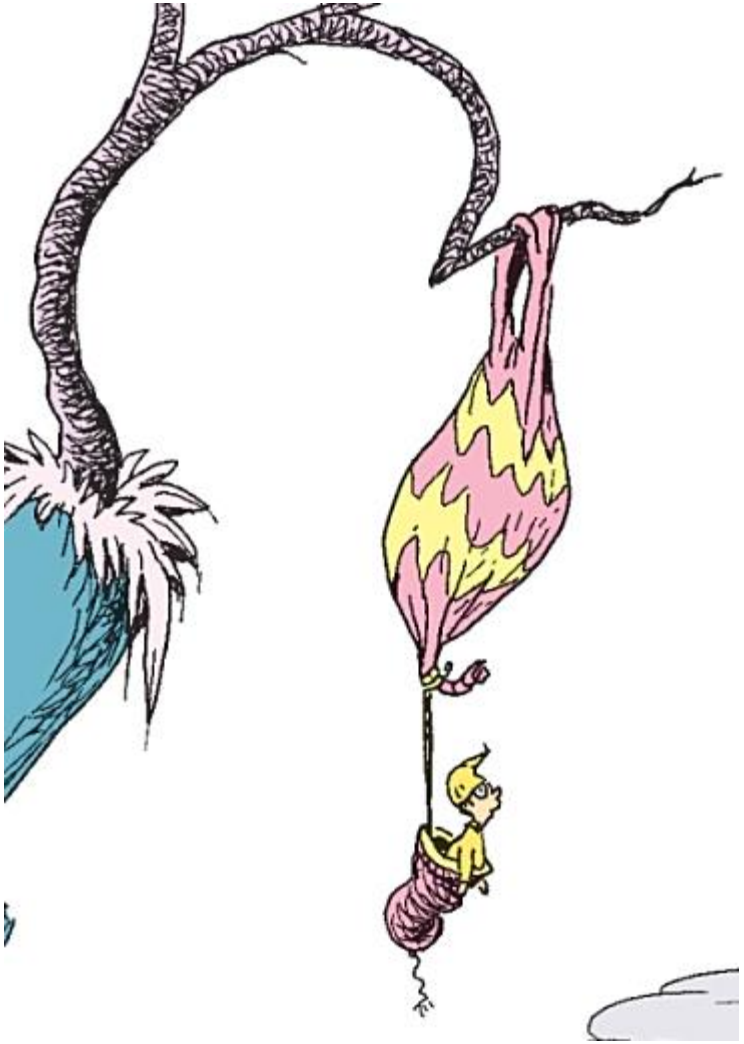
CS 50

March 3, 2026

Programming

Except when they *don't*
Because, sometimes, they *won't*





Common Types of Errors

- **Syntax Errors:** there is something wrong with the structure of the program, and Python doesn't understand it
- **Runtime Errors:** something goes wrong while the program is running (Exceptions)
- **Semantic Errors:** the program runs, but it doesn't do what you want it to do

Example Runtime Errors

- **NameError:** Python doesn't recognize a (variable) name
- **TypeError:** Python can't perform that operation/function on values of that type
- **ValueError:** Python can't perform that operation/function on that particular value
- **IOError:** Python can't find (or can't open) a filename you gave it
- **KeyError:** Python can’t find they key you gave it

What if your code depends on inputs you don't control?

```
def add_ints_from_file(filename):
    s = 0

    file = open(filename, "r")
    for line in file:
        s = s + int(line)
    file.close()

    return s
```

- what if the file doesn't exist?
- what if it does exist but you don't have access permissions?
- what if the file exists and you can open it for reading, but it doesn't contain integers?

What if your code depends on inputs you don't control?

- Best answer: check whether input will work before using it
  - `if str.isdigit(user_in)`
  - `if key in my_dictionary.keys()`
- Alternate answer: try it and crash if an error occurs
- Alternate answer: try it and recover if an error occurs
  - Warning: very inefficient

Exception Handling

- A flexible mechanism for handling errors

```
try:
    # code to execute
except:
    # what to do if there's an error
```

Example: Exception Handling

```
def add_ints_from_file_2(filename):
    s = 0

    try:
        file = open(filename, "r")
        for i in file:
            s = s + int(i)
        file.close()
    except:
        print("An error occurred")
        s = -1

    return s
```

Exercise 1: Exception Handling

- Write a function `return_int` that asks the user to enter an integer. If the user enters an integer, the function returns that integer. If the user does not enter an integer, the function returns 0.

- Use exceptions to handle the case where the user does not enter an integer. Do not use an if statement.

Example: Exception Handling

```
def add_ints_from_file_2(filename):
    s = 0

    try:
        file = open(filename, "r")
        for i in file:
            s = s + int(i)
        file.close()
    except:
        print("An error occurred")
        s = -1

    return s
```

Handling Multiple Exceptions

- A flexible mechanism for handling errors

```
try:
    # code to execute
except <Error1>:
    # what to do if Error1 occurs
except <Error2>:
    # what to do if Error 2 occurs
```

Example: Handling Multiple Exceptions

```
def add_ints_from_file_3(filename):
    s = 0
    try:
        file = open(filename, "r")
        for i in file:
            s = s + int(i)
        file.close()
    except IOError:
        print("problem opening file")
        s = -1
    except ValueError:
        print("file contained non-integer value")
        file.close()
        s = -1
    return s
```

Exercise 2: Handling Multiple Exceptions

- Define a function `list_avg` that takes one parameter `lst` (a list) and returns the average of that list. If the list contains non-numeric values, the function should print a message explaining the error and then return -1. If the list is empty, the function should print an explanation and then return 0.

- Use exceptions to handle the two corner cases. Do not use an if statement in your function definition.

```
>>> 50/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    50/0
    ~~^~
ZeroDivisionError: division by zero
```

Handling the non-exception case

- A flexible mechanism for handling errors

```
try:
    # code to execute
except <Error1>:
    # what to do if Error1 occurs
except <Error2>:
    # what to do if Error 2 occurs
else:
    # additional code if no errors
```

Example: Handling the non-exceptional case

```
def add_ints_from_file_4(filename):
    s = 0
    try:
        file = open(filename, "r")
    except IOError:
        print("problem opening file")
        s = -1
    else:
        for i in file:
            try:
                s = s + int(i)
            except ValueError:
                print("problem with non-integer")
                s = -1
        file.close()
    return s
```

Exercise 3: Exception Handling

```
def f(x,y):
    try:
        try:
            for i in x:
                print(int(i))
            print(x[y])
        except ValueError:
            print(1)
        except TypeError:
            print(2)
        except IndexError:
            print(3)
        else:
            print(4)
        print(5)
    except:
        print(6)
```

- What happens when you evaluate `f("123abc","a")`?
- What happens when you evaluate `f([1,2,3],4)`?
- What happens when you evaluate `f([4,5,6],1)`?
- What happens when you evaluate f({"1":"a", "2":b}, "3")?

Raising Exceptions

- You can use the raise keyword to throw your own exceptions
  - raise Exception("CS50 Exception")
  - raise ValueError("Invalid filename")

Example: Raising Exceptions

```
def exception_v3(filename):
    s = 0
    try:
        file = open(filename, "r")
        for i in file:
            s = s + int(i)
        file.close()
    except IOError:
        raise ValueError("Invalid filename")
    except ValueError:
        file.close()
        raise ValueError("file contained non-int")
    return s
```

Exercise 4: Raising Exceptions

Write a function `return_pos_int_onetry` that asks the user to enter a positive integer. If the user enters a positive integer, the function returns that integer. If the user does not enter a positive integer, the function raises a ValueError.