



CHAPTER 13

FILES AND EXCEPTION HANDLING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING USING PYTHON, LIANG (PEARSON 2013)

MOTIVATIONS

- Data stored in the program are temporary; they are lost when the program terminates. How would you permanently store data created from a program?
- Example:
 - In bioinformatics, the result of a DNA test should be automatically stored for understanding in a crime scene investigation.
- How can our programs work with a large amount of data?
- Example:
 - How could you parse through 1,000,000 tweets related to a major world event to learn which countries it affects the most?
- To permanently store the data created in a program, you need to save them in a **file** on a disk or other permanent storage. The file can be transported and can be read later by other programs. There are two types of files: **text** and **binary**. Text files are essentially strings on disk.



MOTIVATIONS

- When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully?
- Example
 - You are working on Microsoft Word, and you try to open a file that does not exist OR is an incorrectly formatted .doc or .docx file (like someone tampered with it). What should happen?
 - (a) Microsoft crashes
 - (b) Microsoft alerts you of the issue
 - (c) Forget Microsoft, Apple is superior!



INPUT AND OUTPUT

- Input devices



Keyboard



Mouse



Hard drive



Network



Digital camera



Microphone

- Output devices.



Display



Speakers



Hard drive



Network



Printer



MP3 Player

- Goal. Programs that interact with the outside world.

- Programming languages support these interactions
- We use the Operating System (OS) to connect our program to them

WHAT HAVE WE SEEN SO FAR?

- Standard output.

- The OS output stream for text
- By default, standard output is sent to Terminal.
- Example: `print()` goes to standard output.

- Standard input.

- The OS input stream for text
- By default, standard input is received from the Terminal.
- Example: `input()`

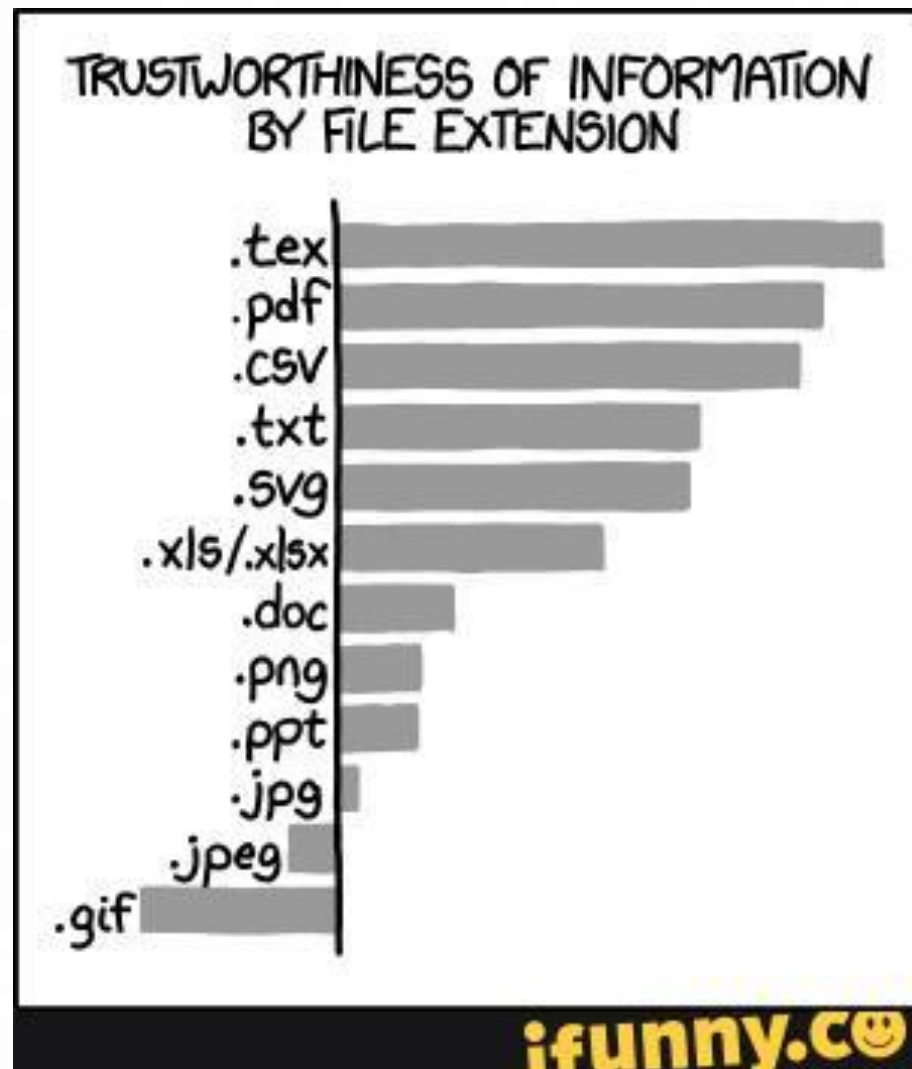
- “Standard Draw.”

- Graphics and GUI libraries
- Output to a window instead of a terminal
- Example: turtle

- EasyGoPiGo3

- Robot actuation and sensing
- Example: light and color sensor

FILE INPUT AND OUTPUT



OPEN A FILE

- How do you write data to a file and read the data back from a file?
- You need to create a file object that is associated with a physical file. This is called opening a file. The syntax for opening a file is as follows:

```
file = open(filename, mode)
```

- Examples:

```
f1 = open("MyOutputFile.txt", 'w')
```

```
f2 = open("MyInputFile.txt", 'r')
```

Mode	Description
'r'	Open a file for reading only.
'w'	Open a file for writing only.
'a'	Open a file for appending data. Data are written to the end of the file.
'rb'	Open a file for reading binary data.
'wb'	Open a file for writing binary data.

EXAMPLE

- Lets write a series of random numbers to a file:

```
import random
file = open("Numbers.txt", 'w')
for i in range(100):
    file.write(str(random.randint(1, 10000)) + '\n')
file.close()
```


EXAMPLE

- Lets read those numbers back and average them:

```
f = open('Numbers.txt', 'r')
line = f.readline()
avg = 0
count = 0
while line != '':
    count += 1
    avg += int(line)
    line = f.readline()
f.close()
print("Average:", avg/count)
```

METHODS OF FILE OBJECTS

file
<code>read([number: int]): str</code>
<code>readline(): str</code>
<code>readlines(): list</code>
<code>write(s: str): None</code>
<code>close(): None</code>

Returns the specified number of characters from the file. If the argument is omitted, the entire remaining contents are read.

Returns the next line of file as a string.

Returns a list of the remaining lines in the file.

Writes the string to the file.

Closes the file.

TESTING FILE EXISTENCE

- You can easily test if a file exists or not:

```
import os.path
if os.path.isfile("Numbers.txt"):
    print("Numbers.txt exists")
```

EXCEPTIONS

- What happens when open is called on a file that does not exist?

Traceback (most recent call last):

```
File ".\TracingTests.py", line 7, in <module>
```

```
    f = open('Number.txt', 'r')
```

```
FileNotFoundError: [Errno 2] No such file or  
directory: 'Number.txt'
```

EXCEPTION HANDLING

- To protect against exceptions, a try-except clause may be used.
- In exception handling, you have to provide an algorithm that takes care of the exception, called the handler.

- Syntax:

```
try:
```

```
    body
```

```
except ExceptionType:
```

```
    handling code
```

EXAMPLE OF EXCEPTION HANDLING

```
try:
    f = open('Numbers.txt', 'r') # Could possibly raise an exception
    line = f.readline()
    avg = 0
    count = 0
    while line != '':
        count += 1
        avg += int(line)
        line = f.readline()
    f.close()
    print("Average:", avg/count)
except FileNotFoundError:
    print("Numbers.txt does not exist.") # Or maybe put in loop to retry?
```

EXCEPTION HANDLING

- A broader syntax exists to handle various exception types. Full syntax:

```
try:  
    body that might raise exception  
except ExceptionType1:  
    code handling exceptions of type ExceptionType1  
...  
except ExceptionTypeN:  
    code handling exceptions of type ExceptionTypeN  
except:  
    code handling any other exception types  
else:  
    code that runs if there is no exception  
finally:  
    code that will run after any excepts or the else
```

RAISING EXCEPTIONS

- The previous code is all about invoking methods that might generate exceptions. In these cases, a try-except clause is appropriate when you know how to handle the code.
 - Imagine, opening a file in Microsoft Word again. The code generates an exception on an illegal file (or poorly formatted one) and allows a user to try again (it doesn't just crash!)
- But how about when we detect an error, but do not know how to handle it?
 - In these cases, we should raise an exception so that the invoker of the method can then decide to handle it.



RAISING EXCEPTIONS

- To raise an exception use the following syntax:

```
raise ExceptionClass("Descriptive message")
```

- Example:

```
class Circle:  
    def __init__(self, r):  
        if r < 0: # Note, we do not know how to  
                # handle the bad input here  
            raise ValueError("Bad radius")  
        self.__radius = r  
    ...
```

NOTE

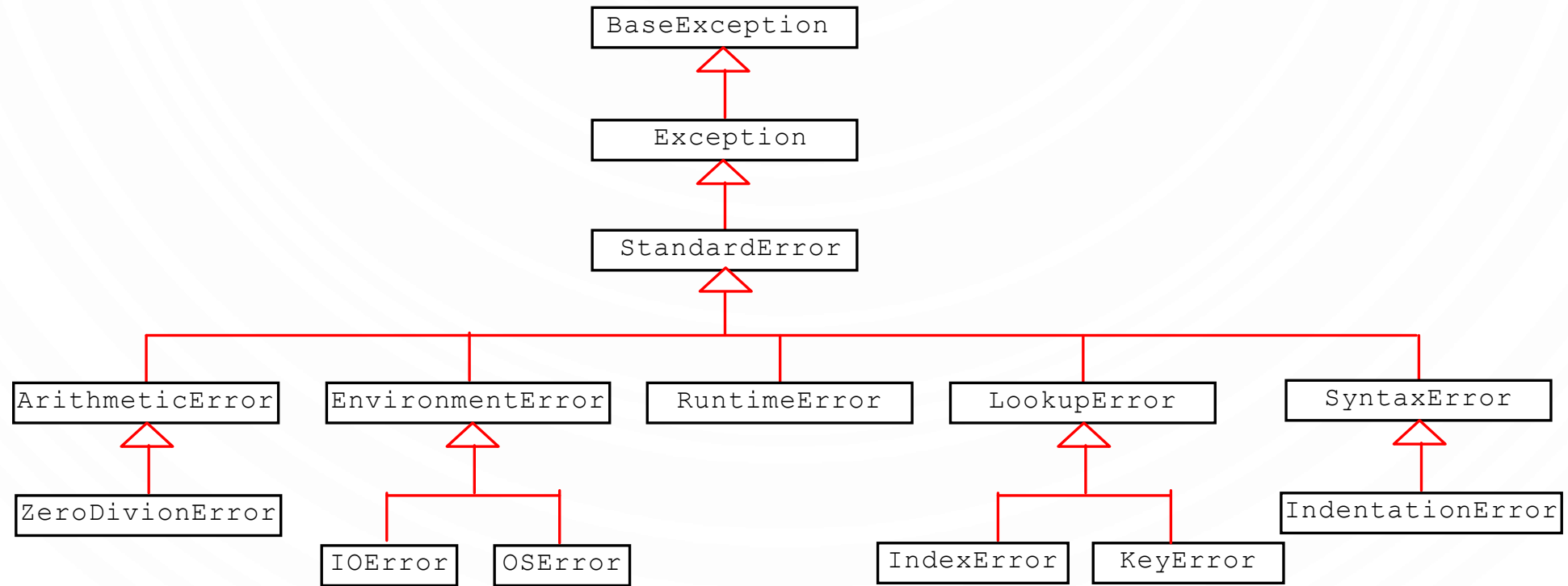
- Raising exceptions "returns" an object. We can access the object in calling code with the following syntax:

```
try:  
    body  
except ExceptionType as exceptionName:  
    handling code
```

- Example:

```
try:  
    f = open('Numbers.txt', 'r')  
except FileNotFoundError as fnfe:  
    print(fnfe)
```

BUILT-IN PYTHON EXCEPTIONS



DEFINING YOUR OWN EXCEPTION CLASS

- You can define your own exception classes, but it requires the use of inheritance. See the book for a detailed example.

- A loose example:

```
class RadiusException(RuntimeError):  
    def __init__(self, radius):  
        super().__init__()  
        self.radius = radius
```

- At the error site:


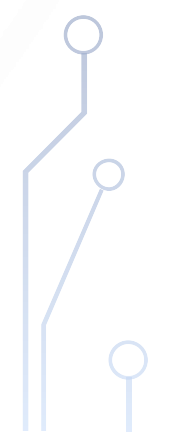
```
if radius < 0:  
    raise RadiusException(radius)
```

- At the exception site:

```
except RadiusException as ex:  
    print("The radius", ex.radius, "is invalid.")
```



CAUTIONS WHEN USING EXCEPTIONS

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.
- 
- 

WHEN TO THROW EXCEPTIONS

- When a variable reaches an unexpected value or algorithm reaches an unexpected state.
- If you want the exception to be processed by its caller, you should create an exception object and raise it.
- However, if you can handle the exception in the method where it occurs, there is no need to throw it, e.g., with if-else statements.



EXERCISE – WORK IN PAIRS/TRIPLETS

EXERCISE

1. Write a program that will generate N random circles, where $N \in [3, 20]$, the center (x, y) points between $[-500, 500]$, and the radius is between $[10, 90]$. Write the circles to a file – first line is N , each line after is the circle defined by x, y , and r
2. Write a program that reads your file and shows it to the user using Turtle. Try to use a random color to show the outline and a different random color to fill the circle.

REMOTE FILES

- When working with the GoPiGo3, any file we write in a python program will be saved on the GoPiGo3 itself.
- We may want to do this for "data-dumps" – a time stamped file with all of the sensor readings, AI decisions, and actuator commands.
- How can we access the files?
 - Remote copy: `scp pi@gopigoXX:~/remoteFile .`
 - Example: `scp pi@gopigo00:~/MyFile.txt .`

EXAMPLE

```
from easygopigo3 import EasyGoPiGo3
robot = EasyGoPiGo3()
distance_sensor = robot.init_distance_sensor()
servo = robot.init_servo()
servo.reset_servo()

f = open("Distances.txt", 'w')
for i in range(36):
    f.write(str(distance_sensor.read_mm()) + '\n')
    robot.turn_degrees(10)
```

COMPETITION



- Form teams and solve the following problem. We will decide who wins tomorrow.
- Make a robot follow a line. Whichever team gets the furthest along the line wins.
 - You can only use the raw data from the line sensor, i.e., the 6 data values.
 - You can assume that you start on the line.
 - You can assume that you will encounter curves and sharp turns up to 90 degrees.
 - The line might be of varying thickness.