

PROGRAMMING WITH PYTHON

By Randal Root

Module 07

In this module, you learn about **creating scripts using custom functions, files, and structured error handling**. You **also** learn a little about creating **advanced GitHub web pages**.

Working with Text files	2
Read Mode	3
Append Mode	4
Reading Data Options	5
Combining Reading and Writing	11
Working with Binary Files	13
LAB 7-1: Working with binary files	14
Structured Error Handling (Try-Except)	14
Using the Exception Class	16
Catching Specific Exceptions	16
Raising Custom Errors	17
Creating Custom Exception Classes	18
Creating Advanced GitHub Pages	19
Creating a Markdown File	19
Formatting the Page	21
Summary	26

Working with Text files

We have already learned how to **write data to a file** using code like Listing 1's. Note that I have "wrapped" the code in a function. By **using a custom function**, I can **customize the docstring** and perform the **open()**, **write()**, and **close()** actions with one **function call**.

```
# ----- #
# Title: Listing 1
# Description: Writing to a file with write()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

# Presentation ----- #
save_data(strData, strFileName)
print('Data Saved!')
```

Listing 1

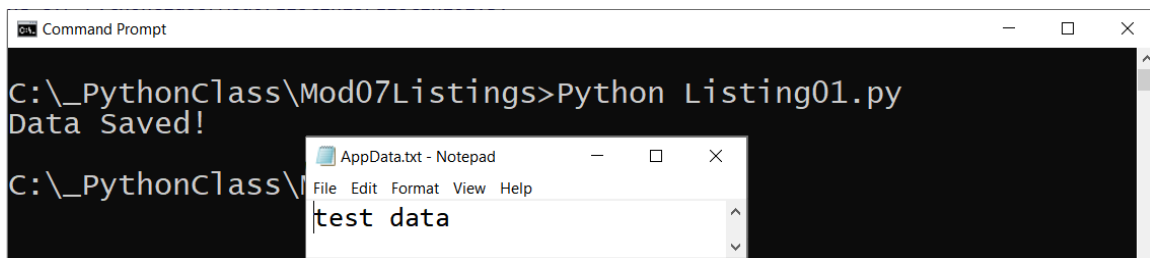


Figure 1. The results of Listing 1

Read Mode

We also learned how data could be **read from a file** using code like Listing 2's.

```
# ----- #
# Title: Listing 2
# Description: Reading from a file with read()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

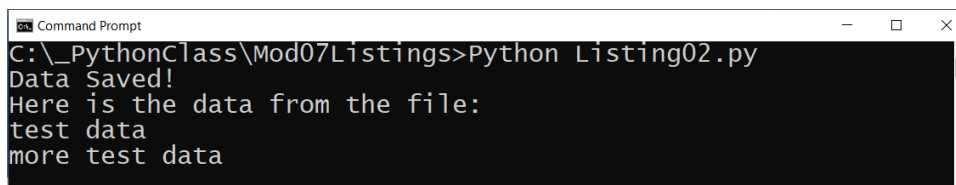
    :param data: (string) with data to save
    :param file_name: (string) with the name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

def read_data(file_name):
    """ Reads all string data from a file

    :param file_name: (string) with name of file
    :return: (string) of data read from the file
    """
    file = open(file_name, "r")
    data = file.read() # read all the data in the file at once
    file.close()
    return data

# Presentation ----- #
save_data(data=strData, file_name=strFileName)
print('Here is the data from the file:')
print(read_data(file_name=strFileName))
print('^ Note the extra blank line was read from the file too! ^')
```

Listing 2



```
Command Prompt
C:\_PythonClass\Mod07Listings>Python Listing02.py
Data Saved!
Here is the data from the file:
test data
more test data
```

Figure 2. The results of Listing 2

Append Mode

And, of course, you can **add data to a new or existing file** using append mode (Listing 3).

```
# ----- #
# Title: Listing 3
# Description: Adding data to a file with append()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

def read_data(file_name):
    """ Reads all string data from a file

    :param file_name: (string) with name of file
    :return: (string) with data read from the file
    """
    file = open(file_name, "r")
    data = file.read() # read all the data in the file at once
    file.close()
    return data

def add_more_data(data, file_name):
    """ Saves string data to a file using APPEND Mode

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "a")
    file.write(data + "\n")
    file.close()

# Presentation ----- #
```

```

save_data(data=strData, file_name=strFileName)
print('Here is the beginning data from the file:')
print(read_data(file_name=strFileName))

add_more_data(data="even more test data", file_name=strFileName)
print('Here is the beginning AND the added data from the file:')
print(read_data(file_name=strFileName))

print('^ Note the extra blank line was read from the file too! ^')

```

Listing 3

```

C:\_PythonClass\Mod07Listings>Python Listing03.py
Here is the beginning data from the file:
test data
more test data

Here is the beginning AND the added data from the file:
test data
more test data
even more test data

^ Note the extra blank line was read from the file too! ^

```

Figure 3. The results of Listing 3

Reading Data Options

Python makes working with files easy, but there may be **some confusion because of** the different ways you can work with files. For instance, there are **several ways to read the data from a file**. Let's look at some common examples.

The readline() function:

Each time you call the readline() method, gets **one line of data and advances to the next line**. Advancing one line at a time is commonly referred to in programming as a **cursor**. Note that since I closed the file, any additional calls to my read_data() function read the same first row of data!

```

# ----- #
# Title: Listing 4
# Description: Reading a single line from a file with readline()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):

```

```

""" Saves string data to a file

:param data: (string) with data to save
:param file_name: (string) with name of file
:return: nothing
"""

file = open(file_name, "w")
file.write(data + "\n")
file.close()

def read_data_row(file_name):
    """ Reads a row of string data from a file

    :param file_name: (string) with name of file
    :return: (string) with one row of data from the file
    """

    file = open(file_name, "r")
    # readline() acts like a "cursor"
    data = file.readline() # read one row of data in the file
    file.close()
    return data

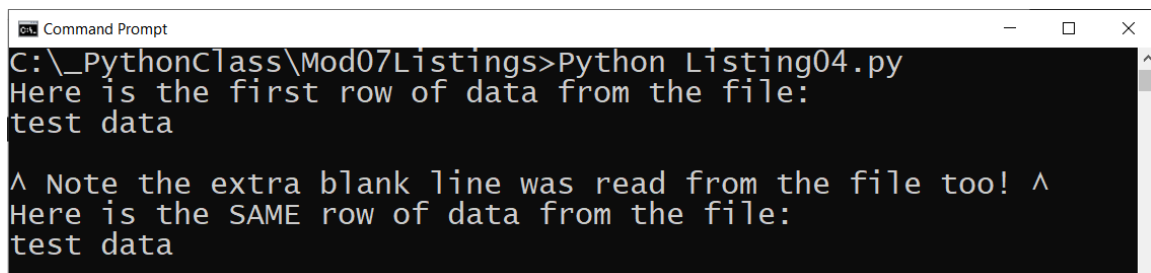
# Presentation ----- #
save_data(data=strData, file_name=strFileName)

print('Here is the first row of data from the file:')
print(read_data_row(file_name=strFileName))
print('^ Note the extra blank line was read from the file too! ^')

print('Here is the SAME row of data from the file:')
print(read_data_row(file_name=strFileName))

```

Listing 4



```

C:\_PythonClass\Mod07Listings>Python Listing04.py
Here is the first row of data from the file:
test data

^ Note the extra blank line was read from the file too! ^
Here is the SAME row of data from the file:
test data

```

Figure 4. The results of Listing 4

Using a while Loop

If you want to get data from **additional lines you must call the readline() method repeatedly**. One way to call the readline() method repeatedly is to use a "while" loop (Listing 5).

```
# ----- #
# Title: Listing 5
# Description: Reading a number of rows from a file with readline()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Data ----- #
strData = 'test data\nmore test data'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

def read_some_data_rows(file_name, number_of_rows):
    """ Reads rows of string data from a file

    :param file_name: (string) with name of file
    :param number_of_rows: (int) with number of rows you want from the file
    :return: (list) with one or more data rows read from the file
    """
    counter = 0
    data = []
    file = open(file_name, "r")
    while counter < number_of_rows:
        data.append([file.readline()]) # APPENDING the data to a list
        counter += 1
    file.close()
    return data # returning the chosen row of data

def read_a_data_row(file_name, row_you_want):
    """ Reads rows of string data from a file

    :param file_name: (string) with the name of file
    :param row_you_want: (int) with the number of the row you want from the
file
    :return: (string) with one or more data rows read from the file
```

```

"""
counter = 0
file = open(file_name, "r")
while counter < row_you_want:
    data = [file.readline()] # REPLACING the data in a list
    counter += 1
file.close()
return data # returning the chosen row of data

# Presentation ----- #
save_data(data=strData, file_name=strFileName)

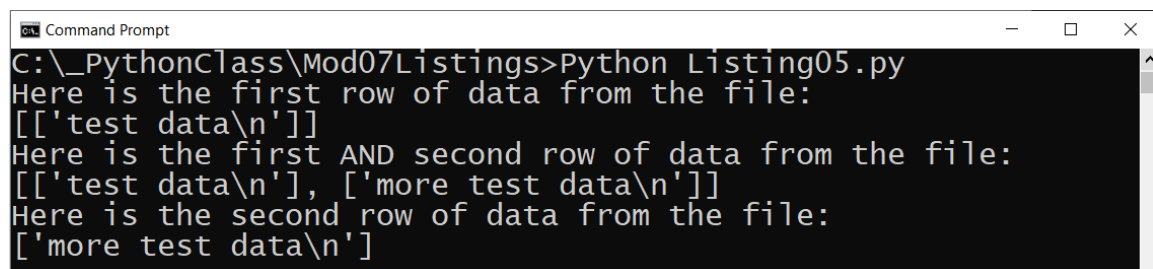
print('Here is the first row of data from the file:')
print(read_some_data_rows(file_name=strFileName, number_of_rows=1)) # new
line character is not included!

print('Here is the first AND second row of data from the file:')
print(read_some_data_rows(file_name=strFileName, number_of_rows=2))

print('Here is the second row of data from the file:')
print(read_a_data_row(file_name=strFileName, row_you_want=2))

```

Listing 5



```

C:\_PythonClass\Mod07Listings>Python Listing05.py
Here is the first row of data from the file:
[['test data\n']]
Here is the first AND second row of data from the file:
[['test data\n'], ['more test data\n']]
Here is the second row of data from the file:
['more test data\n']

```

Figure 5. The results of Listing 5

The readlines() function

Python's readlines() function, **reads all the lines in a file, and returns a list**. The readlines() function is different than the read() function, which reads all the lines in a file and returns a string.

```
# ----- #
# Title: Listing 6
# Description: Reading rows of data from a file with readlines()
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

# Data ----- #
strData = 'ID\nName'
strFileName = 'AppData.txt'
lstData = []

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    file = open(file_name, "w")
    file.write(data + "\n")
    file.close()

def read_file_data_to_list(file_name):
    """ Reads rows of data data from a file into a list

    :param file_name: (string) with name of file
    :return: (list) with rows of data read from the file
    """
    file = open(file_name, "r")
    data = file.readlines() # reads rows of data into a list object
    file.close()
    return data

# Presentation ----- #
save_data(data=strData, file_name=strFileName)

print('Here is the first row of data from the file:')
lstData = read_file_data_to_list(file_name=strFileName)
print(lstData[0].strip()) # used strip() to remove newline
print(lstData[1].strip())
```

Listing 6

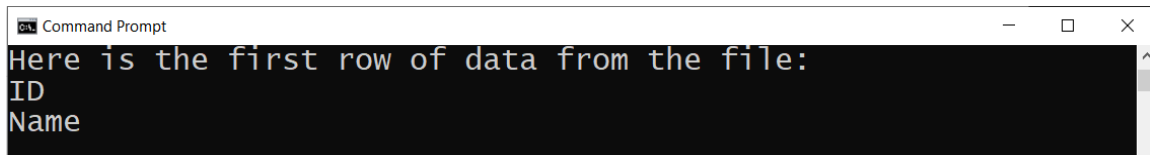


Figure 6. The results of Listing 6

Using a for Loop

Yet another option to **read multiple rows** of data using a "for" loop as shown in Listing 7. One small advantage of using the for loop is that it **automatically closes the file** when it reaches the end of the file's data.

```
# ----- #
# Title: Listing 7
# Description: Reading rows of data from a file with a for loop
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

# Data ----- #
strData = 'ID,Name\n1,Bob Smith'
strFileName = 'AppData.txt'

# Processing ----- #
def save_data(data, file_name):
    """ Saves string data to a file

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :return: nothing
    """
    with open(file_name, "w") as file: # using the WITH option
        file.write(data + "\n")
    # file.close() # with automatically closes the file

def read_file_data_to_list(file_name):
    """ Reads rows of data data from a file into a list

    :param file_name: (string) with name of file
    :return: (list) of data rows read from the file
    """
    data = [] # you must initialize the list variable before you use it
    for row in open(file_name, 'r'):
        data.append(row) # read one row of data in the file per loop
    # automatically closes the file
    return data

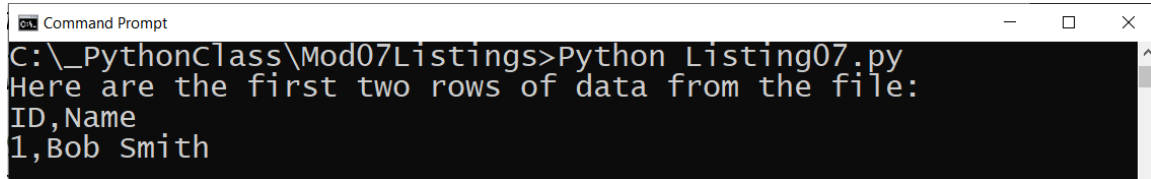
# Presentation ----- #
save_data(data=strData, file_name=strFileName)
```

```

print('Here are the first two rows of data from the file:')
print(read_file_data_to_list(file_name=strFileName)[0].strip())
print(read_file_data_to_list(file_name=strFileName)[1].strip())

```

Listing 7



```

C:\_PythonClass\Mod07Listings>Python Listing07.py
Here are the first two rows of data from the file:
ID,Name
1,Bob Smith

```

Figure 7. The results of Listing 7

Combining Reading and Writing

Combining reading and writing gives you a simple but useful application (Listing 8). For variety, I use the **"with" construct**, which automatically closes the file when the code reaches the end of the "with" block. **For clarity**, at least from the user of my function's viewpoint, I **indicate that the "w" mode overwrites the file contents**.

```

# ----- #
# Title: Listing 8
# Description: Reading and Writing rows of file data
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

# Data ----- #
strFileName = 'AppData.txt'

# Processing ----- #
def manage_file(file_name, mode, data = None):
    """ A custom wrapper function for the standard open() file function

    :param data: (string) with data to save
    :param file_name: (string) with name of file
    :param mode: (string) with name of mode [Write,Overwrite,Read]
    :return: (string) with data or write/append status
    """
    return_data = ''
    if mode.lower() == 'write':
        with open(file_name, "w") as file:
            file.write(data + "\n")
            return_data = 'New data added to file!'
    elif mode.lower() == 'overwrite':
        with open(file_name, "w") as file:
            file.write(data + "\n")
            return_data = 'File overwritten and new data added to file!'
    elif mode.lower() == 'read':
        for row in open(file_name, 'r'):
            return_data += row
    else:

```

```

        return_data = "No matching mode option"
    return return_data

# Presentation ----- #
print("Type in a Customer Id and Name you want to add to the file")
print("(Enter 'Exit' to quit!)")

while(True):
    print('')
    Choose an option
        1 = Show Current Data
        2 = Add New Data
        3 = Create a New or an Overwrite File
        4 = Exit
    print('')
    choice = input('Enter an Option: ')
    if choice == '1':
        print('Here is the current data from the file:')
        print(manage_file(file_name=strFileName, mode='read').strip())
    elif choice == '2':
        new_data = input("Enter the ID and Name (ex. 1,Bob Smith): ")
        manage_file(file_name=strFileName, mode='write', data=new_data)
    elif choice == '3':
        manage_file(file_name=strFileName, mode='overwrite',
data='ID,Name')
    elif choice == "4":
        break
    else:
        print('Please Enter Choice 1,2,3, or 4!')

```

Listing 8

```

C:\_PythonClass\Mod07Listings>Python Listing08.py
Type in a Customer Id and Name you want to add to the file
(Enter 'Exit' to quit!)

    Choose an option
        1 = Show Current Data
        2 = Add New Data
        3 = Create a New or an Overwrite File
        4 = Exit

Enter an Option: 1
Here is the current data from the file:
ID,Name
1,Bob Smith

    Choose an option
        1 = Show Current Data

```

Figure 8. The results of Listing 8

Note: In Listing 8, the file opens and closes each time a user chooses 1, 2, or 3. **A more efficient option is to store and manage data in memory**, then open and close the file only as needed as we did in Module07

Working with Binary Files

Data can be **saved in binary format** instead of just "plain" text. In Python, this technique **called pickling**. Storing data in a binary format can **obscure the file's content and may reduce the file's size**.

Important: While the file's content may be more difficult for humans to read, it **is not encrypted**. So, do not save sensitive data in a binary file and think it is secure!

```
# ----- #
# Title: Listing 9
# Description: A simple example of storing data in a binary file
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

import pickle # This imports code from another code file!

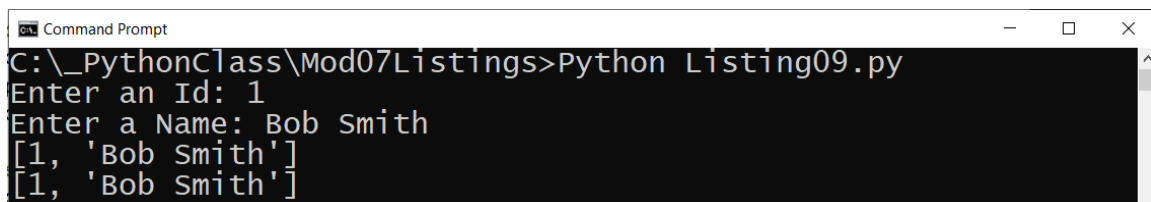
intId = int(input("Enter an Id: "))
strName = str(input("Enter a Name: "))
lstCustomer = [intId, strName]
print(lstCustomer)

# Now we store the data with the pickle.dump method
objFile = open("AppData.dat", "ab")
pickle.dump(lstCustomer, objFile)
objFile.close()

# And, we read the data back with the pickle.load method
objFile = open("AppData.dat", "rb")
objFileData = pickle.load(objFile) #load() only loads one row of data.
objFile.close()

print(objFileData)
```

Listing 9



```
Command Prompt
C:\_PythonClass\Mod07Listings>Python Listing09.py
Enter an Id: 1
Enter a Name: Bob Smith
[1, 'Bob Smith']
[1, 'Bob Smith']
```

Figure 9. The results of Listing 9

Notes:

- The module assignment has you researching information on "Pickling," so we do not cover it in detail here
- We cover how to import code from other files in module 9

In this lab, you create a **function saves data to a binary file**. You will open the file and exam its contents.

- ```

Title: Lab7-1
Description: A simple example of storing data in a binary file
ChangeLog: (Who, When, What)
<YourName>,<1.1.2030>,Created Script

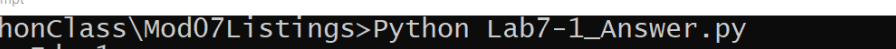
import pickle # This imports code from another code file!

Data -----
strFileName = 'AppData.dat'
lstCustomer = []

Processing -----
def save_data_to_file(file_name, list_of_data):
 pass # TODO: Add code here

def read_data_from_file(file_name):
 pass # TODO: Add code here

Presentation -----
TODO: Get ID and NAME From user, then store it in a List object
TODO: store the list object into a binary file
TODO: Read the data from the file into a new list object and display the
contents
```
- Listing 10

- 
- Command Prompt
- ```
C:\_PythonClass\Mod07Listings>Python Lab7-1_Answer.py
Enter an Id: 1
Enter a Name: aaa
[1, 'aaa']

C:\_PythonClass\Mod07Listings>
```
- AppData.dat - Notepad
- File Edit Format View Help
- €|]q (K|X| aaaq|e.€|]q (K|X| aaaq|e.

Figure 10. The results of Lab 7-1

When you are programming, you fix your bugs immediately and make sure the code runs smoothly. However, it often happens that other **people introduce new bugs when they use your program**. For example, they may change the name of a data file,

causing the file not to be found, or input data that does not fit well with your program's design.

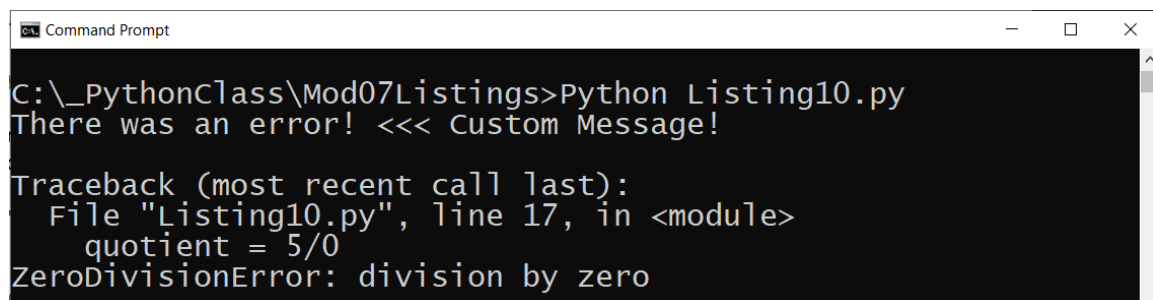
You can trap these errors in your programs using a try-except block of code. Doing so **allows you to customize how your program handles errors instead of just letting Python do that for you.** It is a good idea to **add a try-except block** to your programs **whenever you think human interaction might cause a problem** (Listing 11).

```
# ----- #
# Title: Listing 11
# Description: A simple try-catch demo
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

# Using try
try:
    quotient = 5/0
    print(quotient)
except:
    print("There was an error! <<< Custom Message!\n")

# Now, without try
quotient = 5/0
print('never gets to this line')
```

Listing 11



```
Command Prompt
C:\_PythonClass\Mod07Listings>Python Listing10.py
There was an error! <<< Custom Message!

Traceback (most recent call last):
  File "Listing10.py", line 17, in <module>
    quotient = 5/0
ZeroDivisionError: division by zero
```

Figure 11. The results of Listing 11

Note: The **standard error messages may feel too technical** for the people who use your programs. Using try-except blocks allow you to **create more user-friendly messages.**

Using the Exception Class

"Exception" is a built-in python **class used to hold information about an error**. Python **automatically creates an Exception object when an error occurs**. The Exception object automatically fills **with information about the error** that caused the exception.

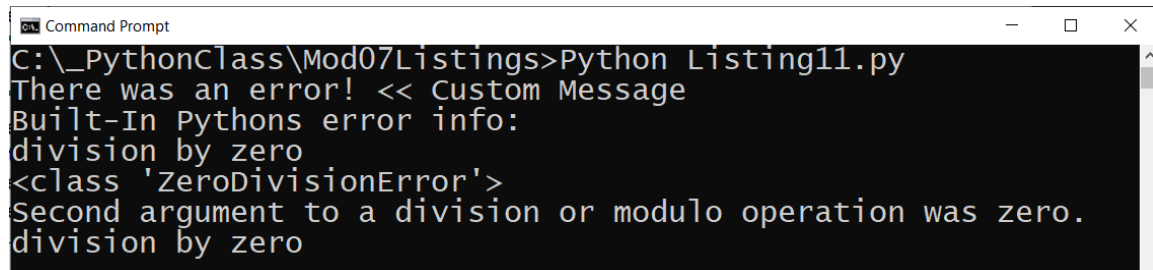
You can **capture the Exception object in the except section** of a try-except block and extract the error messages (Listing 12).

```
# ----- #
# Title: Listing 12
# Description: A try-catch with better error messages
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

try:
    quotient = 5/0
    print(quotient)
except Exception as e:
    print("There was an error! << Custom Message")

    print("Built-In Python's error info: ")
    print(e)
    print(type(e))
    print(e.__doc__)
    print(e.__str__())
```

Listing 12



```
Command Prompt
C:\_PythonClass\Mod07Listings>Python Listing11.py
There was an error! << Custom Message
Built-In Python's error info:
division by zero
<class 'ZeroDivisionError'>
Second argument to a division or modulo operation was zero.
division by zero
```

Figure 12. The results of Listing 12

Catching Specific Exceptions

The Exception class can catch any type of error, but you can **catch specific errors using more specific exception classes** (Listing 13).

```
# ----- #
# Title: Listing 13
# Description: A try-catch with multiple error messages
# https://docs.python.org/3/Library/exceptions.html#builtin-exceptions
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #
```

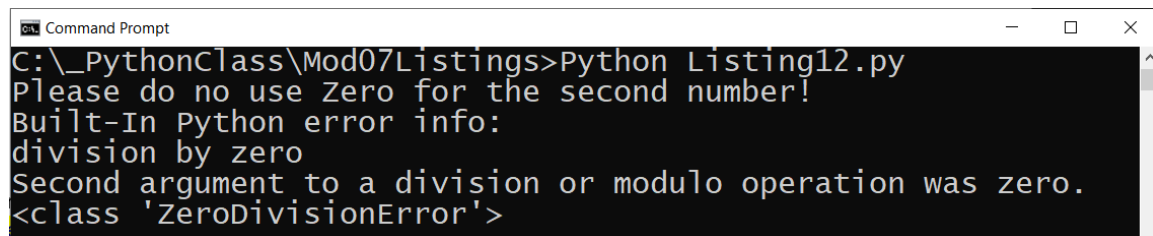


```

try:
    quotient = 5/0
    f = open('SomeFile.txt', 'r') # the read plus option gives an error if
    # file does not exist
    f.write(quotient) # causes an error if the file does not exist
except ZeroDivisionError as e:
    print("Please do no use Zero for the second number!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
except FileNotFoundError as e:
    print("Text file must exist before running this script!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
except Exception as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')

```

Listing 13



```

C:\_PythonClass\Mod07Listings>Python Listing12.py
Please do no use Zero for the second number!
Built-In Python error info:
division by zero
Second argument to a division or modulo operation was zero.
<class 'ZeroDivisionError'>

```

Figure 13. The results of Listing 13

Note: You can find a list of Python's exception classes on this webpage:

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

Raising Custom Errors

Python automatically generates errors based on conditions defined by the Python Runtime. However, **you can also "raise" errors based on custom conditions** (Listing 14).

```

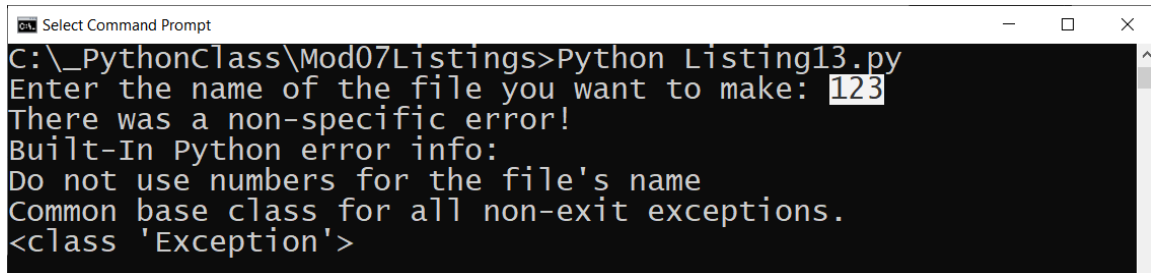
# ----- #
# Title: Listing 14
# Description: A try-catch with manually raised errors
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

try:
    new_file_name = input("Enter the name of the file you want to make: ")
    if new_file_name.isnumeric():
        raise Exception('Do not use numbers for the file\'s name')
except Exception as e:
    print("There was a non-specific error!")

```

```
print("Built-In Python error info: ")
print(e, e.__doc__, type(e), sep='\n')
```

Listing 14



```
Select Command Prompt
C:\_PythonClass\Mod07Listings>Python Listing13.py
Enter the name of the file you want to make: 123
There was a non-specific error!
Built-In Python error info:
Do not use numbers for the file's name
Common base class for all non-exit exceptions.
<class 'Exception'>
```

Figure 14. The results of Listing 14

Creating Custom Exception Classes

In **Figure 13**, you see the message, "**Common base class for all non-exit exceptions.**" This message is **coming from the Exception class's docstring**. It tells an experienced programmer that the **Exception class is a "base" class** that can be used to create "derived" classes. **Derived classes "inherit" data and functions from the base class that can be replaced and customized.**

Listing 15 shows **two classes that derive code from the Exception class** and **replace the "__str__()" function** with a custom message.

Note: We cover why we need to use the keyword "self" and do not use the directive "@staticmethod" in module 8.

```
# ----- #
# Title: Listing 15
# Description: A try-catch with manually raised errors
#              using custom error classes
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030, Created Script
# ----- #

class CustomError(Exception):
    """ Some custom error info in the DocString """
    def __str__(self):
        return 'Some custom error message'

class FileNotTXTError(Exception):
    """ File extension must end with txt to indicate it is a text file """
    def __str__(self):
        return 'File extension not txt'
```

```

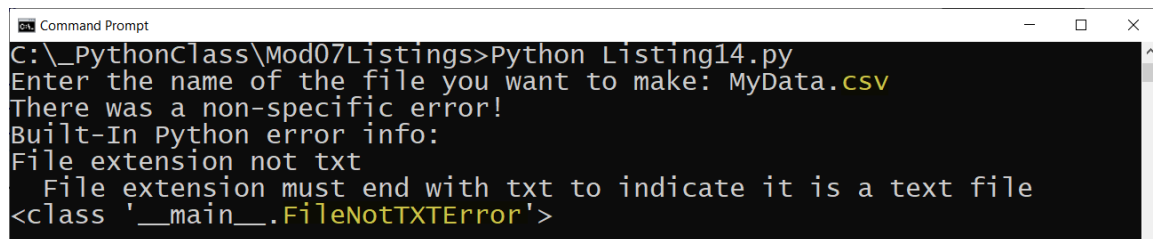
try:
    new_file_name = input("Enter the name of the file you want to make: ")
    if new_file_name.isnumeric():
        raise Exception('Do not use numbers for the file\'s name')
    elif new_file_name.endswith('txt') == False:
        raise FileNotFoundError()
    else:
        raise CustomError()

except Exception as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')

# TODO: add more except sections to capture the specific derived exceptions!

```

Listing 15



```

C:\_PythonClass\Mod07Listings>Python Listing14.py
Enter the name of the file you want to make: MyData.csv
There was a non-specific error!
Built-In Python error info:
File extension not txt
File extension must end with txt to indicate it is a text file
<class '__main__.FileNotTXTError'>

```

Figure 15. The results of Listing 15

Creating Advanced GitHub Pages

In this module's assignment, you create a GitHub webpage that describes how Python's error handling and pickling work. This web page should be **similar to the Word documents you created in previous assignments**. To create your page you need to know some basic commands in a language called "markdown." The markdown code is converted into a static HTML page using a conversion processing application called "Jekyll."

"GitHub combines a syntax for formatting text called GitHub Flavored Markdown with a few unique writing features.

Markdown is an easy-to-read, easy-to-write syntax for formatting plain text.

We've added some custom functionality to create GitHub Flavored Markdown, used to format prose and code across our site." (<https://help.github.com/en/github/writing-on-github/about-writing-and-formatting-on-github>, 2019)

Creating a Markdown File

To demonstrate an example, I create a new GitHub repository called "ITFnd100-Mod07" as shown in Figure 16.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

The screenshot shows the GitHub 'Create a new repository' form. At the top, there's a section for 'Owner' (rootrUW) and 'Repository name' (ITFnd100-Mod07). Below this is a note about repository names being short and memorable. The 'Description' field contains 'Files for module 7'. There are two radio buttons for visibility: 'Public' (selected) and 'Private'. A checkbox 'Initialize this repository with a README' is checked. At the bottom, there are dropdowns for 'Add .gitignore: None' and 'Add a license: None', and a green 'Create repository' button.

Figure 16. Creating a new GitHub repository

Next, I **create a new "docs" folder with a file called "index.md"** inside of it. When I do, I need to **type or paste in some text** for the new file before it is created in the folder. In Figure 17, I have typed in some **simple markdown commands to format my document**. Once I have at least some text in the file I can create the folder and file on my repository.

The screenshot shows the GitHub repository page for 'rootrUW / ITFnd100-Mod07'. The 'Code' tab is selected. Below the repository name, there's a path 'ITFnd100-Mod07 / docs /' followed by a text input field containing 'index.md' and a 'Cancel' button. Below this, there's a section for 'Edit new file' with a 'Preview' tab. The 'Edit' tab is active, showing a markdown file with the following content:

```
1 # Title
2 *RRoot, 1.1.2030*
3
4 ## Introduction
5 Lorem ipsum is a made-up language. It is used as a placeholder for actual text and is
6 "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ips
7 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incidi
8
9 ## Topic 1
10 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incidi
11
```

Figure 17. Creating a new Index.md file in the docs folder

Formatting the Page

I use the "Preview" tab to see what my new markdown file looks like each time I modify the text (Figure 18). The look changes based on what markdown commands I use. While there are lots of commands available, let's **focus on the ones you need for the assignment**.

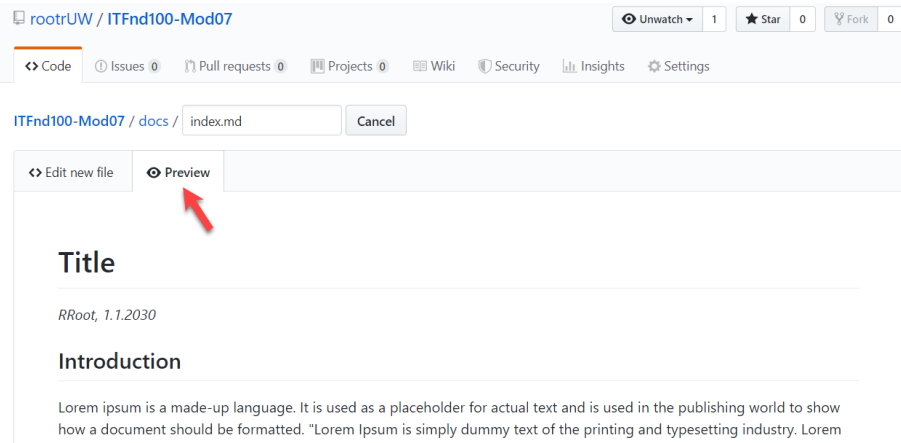


Figure 18. Previewing a GitHub page

Creating Text Headers

The hash, "**#**" **symbol, indicates a header**. You use one or more hash symbols to define the level of the header. Oddly, the more hash marks you use, the smaller the header size (Figure 19).

```
# Title
## Introduction
## Topic 1
### Subtopic
## Summary
```

Listing 16

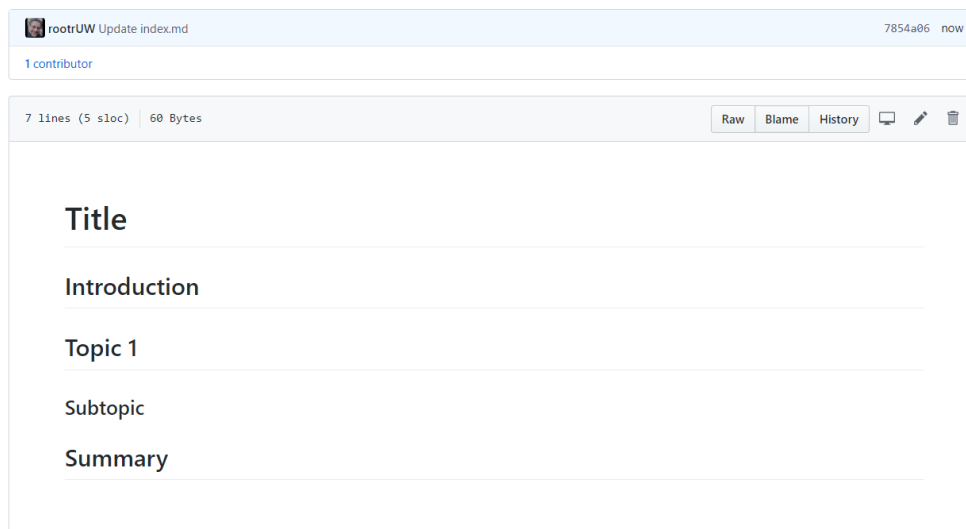


Figure 19. The results of Listing 16

Formatting Text

You can make text **bold using two Astrid "***" symbols** or *italic using a single Astrid*.

```
**Dev:** *RRoot*  
**Date:** *1.1.2030*
```

Listing 17

Note: There are **two invisible space characters after the text *"*RRoot*" that force a newline into the page***. Newlines can be tricky in this language, and you may need to experiment to style the page to your liking.

Title

Dev: *RRoot*
Date: *1.1.2030*

Figure 20

Adding Code Samples

To add some **sample code** to the page, you **use the backtick (`) symbol**, as shown in Listing 18.

```
# Title  
**Dev:** *RRoot*  
**Date:** *1.1.2030*
```

```
## Structured Error Handling (Try-Except)
```

When you are programming, you fix your bugs immediately and make sure the code runs smoothly. However, it often happens that other people introduce new bugs when they use your program.

```
### Raising Custom Errors
```

Python automatically generates errors based on conditions defined by the Python Runtime. However, you can also "raise" errors based on custom conditions (Listing 13).

```
'''  
# ----- #  
# Title: Listing 13  
# Description: A try-catch with manually raised errors  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ----- #
```

```
try:
```

```
    new_file_name = input("Enter the name of the file you want to make: ")  
    if new_file_name.isnumeric():  
        raise Exception('Do not use numbers for the file\'s name')
```

```
except Exception as e:
```

```
    print("There was a non-specific error!")  
    print("Built-In Python error info: ")
```

```
print(e, e.__doc__, type(e), sep='\n')
```

Listing 13

Listing 18

29 lines (24 sloc) | 1.05 KB

RawBlameHistory

Title

Dev: RRoot
Date: 1.1.2030

Structured Error Handling (Try-Except)

When you are programming, you fix your bugs immediately and make sure the code runs smoothly. However, it often happens that other people introduce new bugs when they use your program.

Raising Custom Errors

Python automatically generates errors based on conditions defined by the Python Runtime. However, you can also "raise" errors based on custom conditions (Listing 13).

```
# ----- #
# Title: Listing 13
# Description: A try-catch with manually raised errors
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ----- #

try:
    new_file_name = input("Enter the name of the file you want to make: ")
    if new_file_name.isnumeric():
        raise Exception('Do not use numbers for the file\'s name')
except Exception as e:
    print("There was a non-specific error!")
    print("Built-In Python error info: ")
    print(e, e.__doc__, type(e), sep='\n')
```

Listing 13

Figure 21. The results of Listing 18

Tip: The keyboard key for the **backtick symbol** (`) is shared with the tilde symbol (~) above the **Tab** key on both a Windows and Mac keyboard.

Adding Images

To **add an image** to a page, perform the following steps.

1. **Save image to your computer's hard drive.** In MS Word, you do this by right-clicking the image and using the "Save as Picture..." option of the context menu.

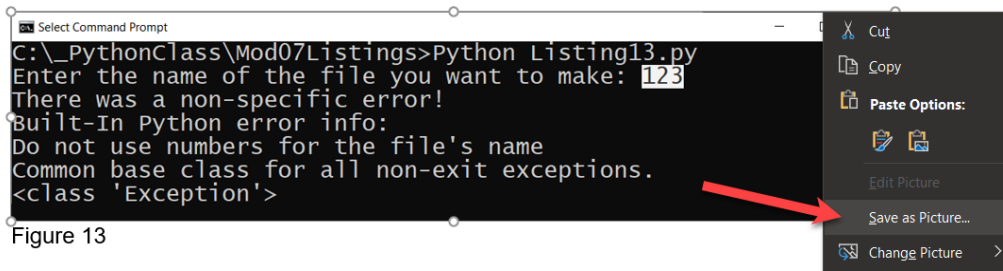


Figure 22. Saving a image from Word to a drive

2. **Upload the image to your GitHub** repository's "docs" folder (or a subfolder if you wish to be more organized).

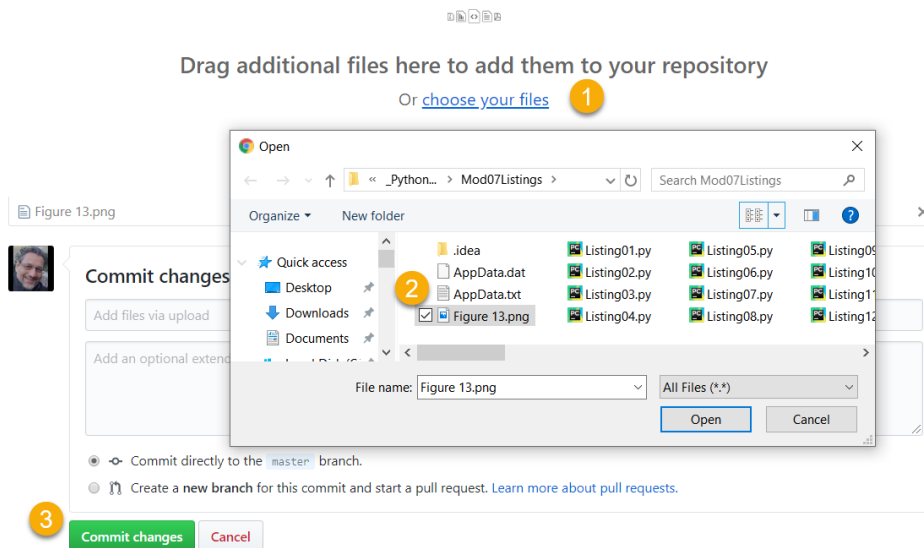


Figure 23. Uploading the saved image file to GitHub

3. **Copy the web address for the file**, by locating your file on the GitHub page, then right-clicking it to access the context menu. From there, use the "Copy link address" option, or an equivalent one if you are using a browser other than Chrome.

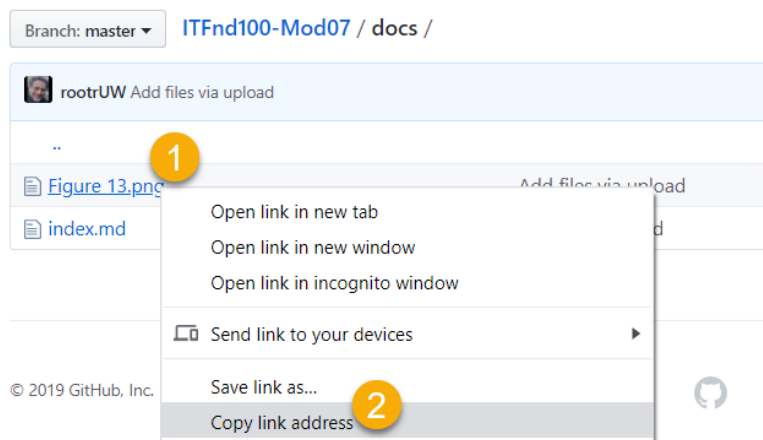


Figure 24. Copying the file's URL on GitHub with a Chrome browser

4. **Link your image to your page using an image link code.** The syntax for the image link is:

```
![alt text](web address "tooltip text")
```

Here is an example:

```
print("Built-In Python error info: ")
print(e, e.__doc__, type(e), sep='\n')
...
#### Listing 13
```

```
![Results of Listing 13](Figure13.png "Results of Listing 13")####
Figure 13. The results of Listing 13
```

The **first part of command** indicates the **alternate text** use by screen readers. The **second** part of the command indicates the **URL to the file**. The **third** part of the command indicates the text you want a **tool tip** to display. Figure 24 shows the results of the command.

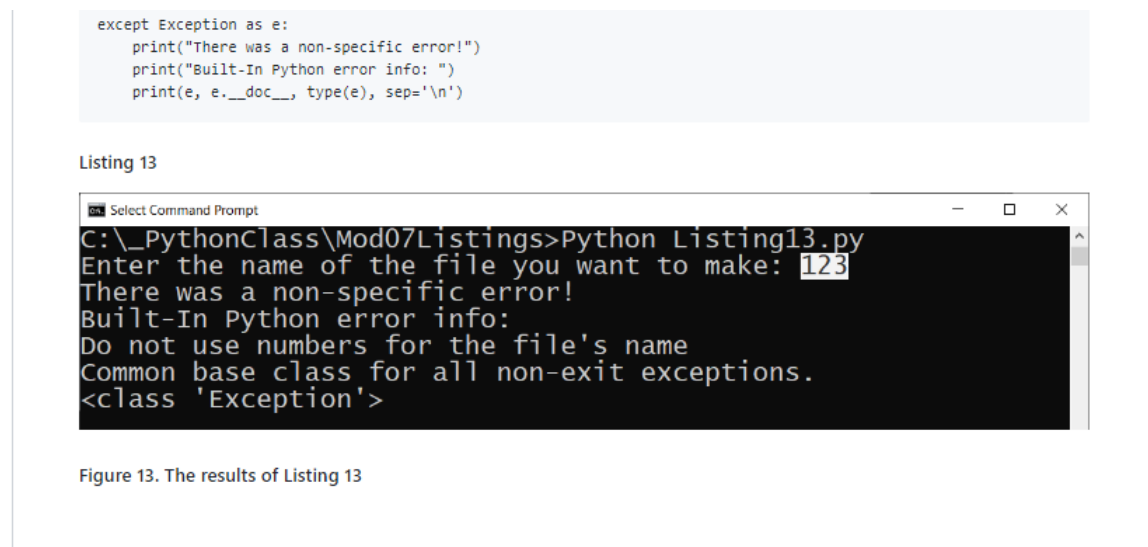


Figure 25. The results of using a Markdown image link on a GitHub page

Important: It is best to use a relative path for files that are in your folder, and only use the "hard coded" physical path for files outside of your website.

Learning More

There is much information on the Internet about the Markdown language, but you **should find all you need for this course on this one webpage:**

<https://help.github.com/en/github/writing-on-github/basic-writing-and-formatting-syntax>

Important: Learning to use Markdown and Jekyll could well be the topic of a complete course, but in this course, **you do NOT need to know much about Markdown**

programming. **Please use only the basics shown in this module** instead of more advanced features **and do not worry about getting the format perfect!**

Summary

In this module, we looked at how to use custom functions and try-except blocks to organize file management code and provided custom error handling. We also looked at ways you can improve your GitHub webpages to look more professional.

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

- What are the benefits of putting built-in Python command into functions?
- What are the benefits of using structured error handling?
- What are the differences between a text file and a binary file?
- How is the Exception class used?
- How do you "derive" a new class from the Exception class?
- When might you create a class derived from the Exception class?
- What is the Markdown language?
- How do you use Markdown on a GitHub webpage?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.