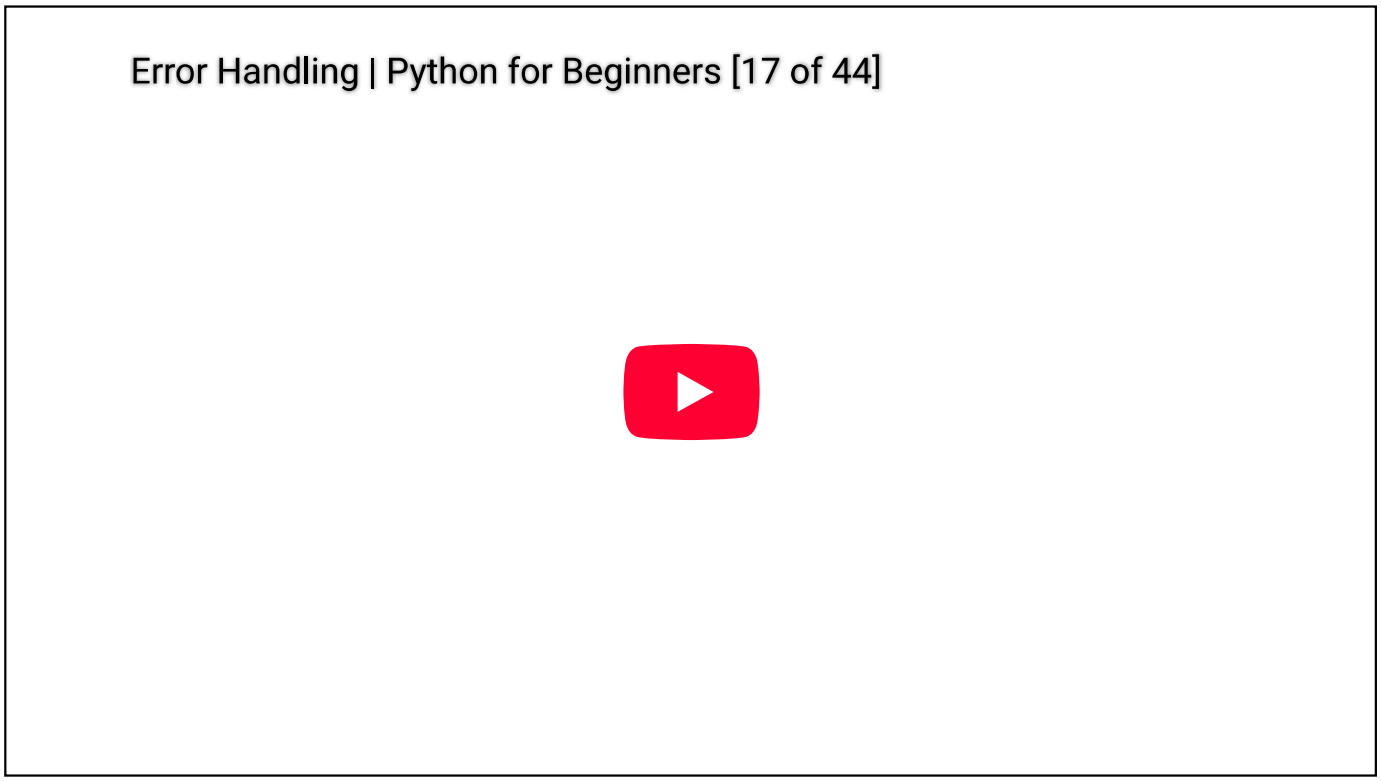
W05 Learning Activity (2 of 2): Exceptions

Errors and exceptional situations sometimes occur while a program is running. Such errors include a program attempting to read from a file that doesn't exist, a connection error when connecting to a server on a network, data that cannot be found on a server, and calculations that produce undefined results. A well written program doesn't crash when an error occurs but instead handles errors in a graceful manner that may include adjusting to an error, printing an error message for the user to see, and saving an error message to a log file. During this lesson, you will learn to write code that handles errors that may occur while your Python program is running.

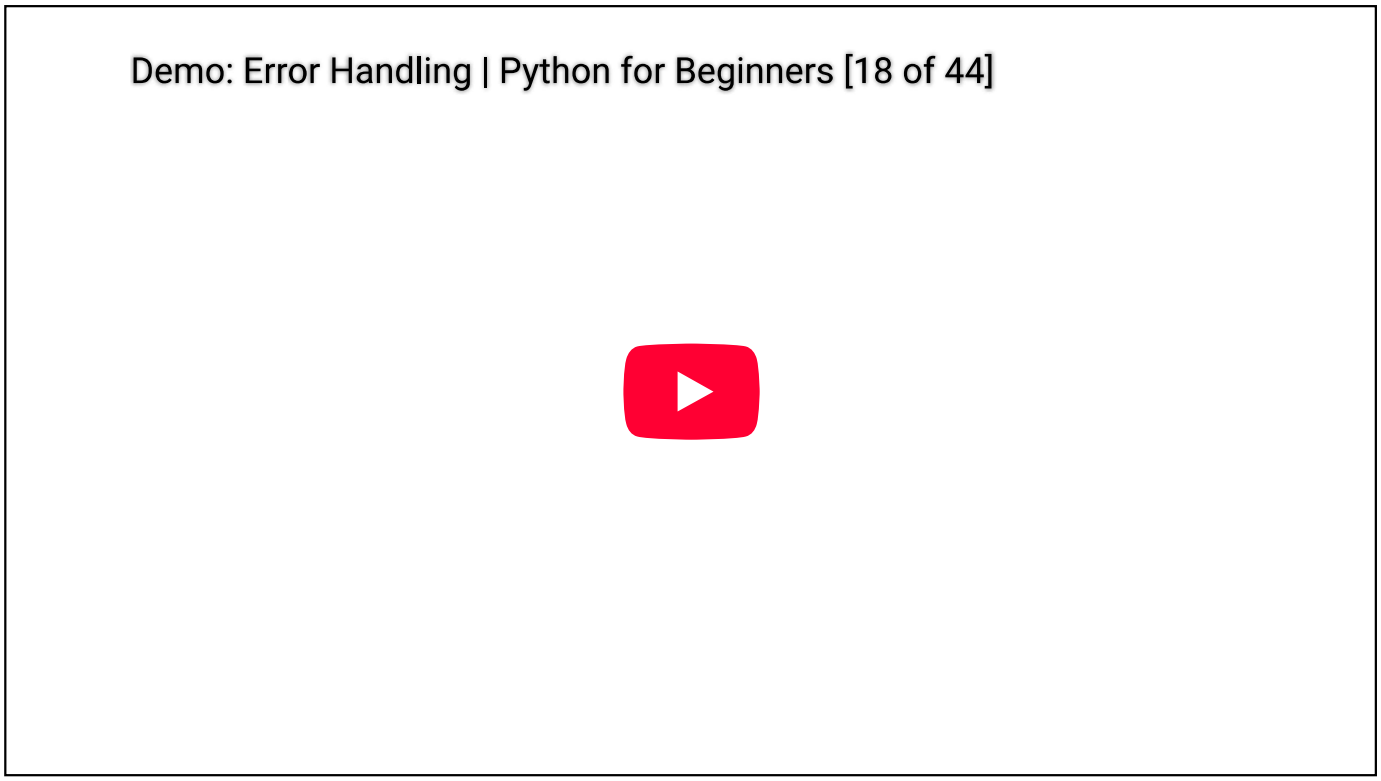
Videos

Watch these two videos from Microsoft about error handling.

1. Error Handling Concepts (13 minutes)



2. Error Handling Demonstration (4 minutes)



Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is an Exception?

An *exception* is a relatively rare event that sometimes occurs while a program is running. For example, an exception occurs when a Python program tries to open a file for reading but that file doesn't exist. There are many different built-in exceptions that may occur while a Python program is running.

When an exceptional event occurs, a Python function *raises* an exception which may be handled by code at another location in the executing Python program. The Python keyword to raise an exception is **`raise`**. Normally, you will not need to write code to raise an exception because the built-in functions, such as **`open`**, **`int`**, and **`float`**, will raise an exception when necessary. You will need to write code in your programs to handle exceptions.

How to Handle an Exception

The Python keywords to handle exceptions are **`try`**, **`except`**, **`else`**, and **`finally`**. The following example code contains the outline of a complete try-except-else-finally block. Read the code and its comments carefully to understand the correct syntax and organization of a try-except-else-finally block.

```
1# Example 1
2try:
3# Write normal code here. This block must include
4# code that falls into two groups:
5# 1. Code that may cause an exception to be raised
6# 2. Code that depends on the results of the code
7# in the first group
8except ZeroDivisionError as zero_div_err:
9# Code that the computer executes if the code in the try
10# block caused a function to raise a ZeroDivisionError.
11except ValueError as val_err:
12# Code that the computer executes if the code in the
13# try block caused a function to raise a ValueError.
14except (TypeError, KeyError, IndexError) as error:
15# Code that the computer executes if the code in the
16# try block caused a function to raise a TypeError,
17# KeyError, or IndexError.
18except Exception as excep:
19# Code that the computer executes if the code in the try
20# block caused a function to raise any exception that
21# was not handled by one of the previous except blocks.
22except:
23# Code that the computer executes if the code in the
24# try block caused a function to raise anything that
25# was not handled by one of the previous except blocks.
26else:
27# Code that the computer executes after the code
28# in the try block if the code in the try block
29# did not cause any function to raise an exception.
30finally:
31# Code that the computer executes after all the other
32# code in try, except, and else blocks regardless of
33# whether an exception was raised or not.
```

As shown in example 1 above, when we want to write code that will handle exceptions, we first write a **`try`** block, and we put into that **`try`** block the normal code that might cause an exception. Then we write **`except`** blocks to handle the exceptions. Each **`except`** block may handle one type of exception like the code at line 8:

```
except ZeroDivisionError as zero_div_err:
```

Or each **`except`** block may handle several types of exceptions, like the code at line 14:

```
except (TypeError, KeyError, IndexError) as error:
```

Or one **except** block may handle all possible types of exceptions, like the code at line 18:

```
except Exception as excep:
```

Or a bare **except** block may handle anything that can be raised, including **SystemExit**, **KeyboardInterrupt** and **GeneratorExit**, like the code at line 22:

```
except:
```

The Python programming language requires us to order **except** blocks from most specific at the top to least specific (most general) at the bottom.

In a Python program, it is usually a bad idea to write a bare **except** block (line 22) because a bare except block will handle **KeyboardInterrupt** and **SystemExit**. Writing code to handle **KeyboardInterrupt**, including a bare except block, may make it difficult for a user to quit your program. The most common reason to write **try** and **except** blocks is to write code that will enable a program to recover from an error. Writing recovery code is easiest in an **except** block that includes the exception type like the **except** blocks at lines 8, 11, and 14. This is why professional programmers rarely write bare except blocks.

As shown at line 26 in example 1 above, following the **except** blocks, a programmer may write an optional **else** block which the computer will execute if the **try** block does not raise any exceptions. However, it is uncommon to write an **else** block for **try** and **except** blocks because any code that could be written in an **else** block of **try** and **except** could also be written at the end of the **try** block. Professional programmers almost never write an **else** block for **try** and **except** blocks.

As shown at line 30 in example 1 above, at the end of the exception handling code, a programmer may write an optional **finally** block. The **finally** block contains code that the computer executes after all the other code in the **try**, **except**, and **else** blocks regardless of whether an exception was raised or not. The code in the **finally** block usually contains “clean up” code that frees resources that the code in the **try** block used. For example, if the code in the **try** block opens a file, the code in the **finally** block could close that file. In CSE 111, you won’t need to write a **finally** block.

To summarize, in CSE 111 you will need to write **`try`** and **`except`** blocks to handle exceptions. You will not need to write bare **`except`** blocks, **`else`** blocks, or **`finally`** blocks to handle exceptions.

Common Exception Types

There are many different types of built-in exceptions that may occur while a Python program is running. This section shows how seven types of exceptions may occur.

TypeError

The computer raises a **`TypeError`** when the code that calls a function passes an argument with the wrong data type. The code in example 2 attempts to pass a string to the **`round`** function. This causes the computer to raise a **`TypeError`** because the **`round`** function cannot round a string to an integer. It can round only a number to an integer. The output below example 2 shows that the computer raised a **`TypeError`**.

```
# Example 2
def main():
  try:
    text = input("Please enter a number: ")
    integer = round(text)
    print(integer)
  except TypeError as type_err:
    print(type_err)
if __name__ == "__main__":
  main()
```

```
> python type_error.py
Please enter a number: 25.7
type str doesn't define __round__ method
```

ValueError

The computer raises a **`ValueError`** when the code that calls a function passes an argument with the correct data type but with an invalid value. A common event that causes the computer to raise a **`ValueError`** is when the **`int`** function or **`float`** function tries to convert a string to a number but the string contains characters that are not digits. The code in example 3 and its output show a **`ValueError`**.

```
# Example 3
def main():
  try:
    number = float(input("Please enter a number: "))
    print(number)
  except ValueError as val_err:
    print(val_err)
if __name__ == "__main__":
  main()
```

```
> python value_error.py
Please enter a number: 45.7u
could not convert string to float: '45.7u'
```

ZeroDivisionError

The computer raises a **ZeroDivisionError** when a program attempts to divide a number by zero (0) as shown in example 4 and its output.

```
# Example 4
def main():
  try:
    players = int(input("Enter the number of players: "))
    teams = int(input("Enter the number of teams: "))
    players_per_team = players / teams
    print(f"Each team should have {players_per_team} players")
  except ZeroDivisionError as zero_div_err:
    print(zero_div_err)
if __name__ == "__main__":
  main()
```

```
> python zero_div_error.py
Enter the number of players: 20
Enter the number of teams: 0
division by zero
```

IndexError

Recall from week 4 that each element in a list is stored at a unique index and that an index is always an integer. If we write code that tries to use an index that doesn't exist in a list, when the computer executes that code, the computer will raise an

IndexError. The program in example 5 creates a list that contains three surnames. Then the program attempts to change the surname at index 3. Of course, the list contains only three elements, and the index of the last element is 2, so the statement fails and causes the computer to raise an **IndexError**.

```
# Example 5
def main():
  try:
    # Create a list that contains three family names.
    surnames = ["Smith", "Lopez", "Marsh"]
    # Attempt to change the surname at index 3. Because
    # there are only three names in the surnames list and
    # therefore the last index is 2, this statement will
    # fail and cause the computer to raise an IndexError.
    surnames[3] = "Olsen"
  except IndexError as index_err:
    print(index_err)
if __name__ == "__main__":
main()
```

```
> python index_error_write.py
list assignment index out of range
```

The program in example 6 is similar to example 5, and both programs cause the computer to raise an **IndexError**. The program in example 6 creates a list that contains three surnames. Then the program attempts to print the surname at index 3. Of course, this statement fails because the list contains only three elements, and the index of the last element is 2.

```
# Example 6
def main():
  try:
    # Create a list that contains three family names.
    surnames = ["Smith", "Lopez", "Marsh"]
    # Attempt to print the surname at index 3. Because
    # there are only three names in the surnames list and
    # therefore the last index is 2, this statement will
    # fail and cause the computer to raise an IndexError.
    print(surnames[3])
  except IndexError as index_err:
    print(index_err)
if __name__ == "__main__":
```

```
14  main()
```

```
> python index_error_read.py
list index out of range
```

KeyError

As shown in example 7, if we write code that attempts to find a key in a dictionary and that key doesn't exist in the dictionary, then the computer will raise a **KeyError**.

```
1# Example 7
2def main():
3  try:
4    # Create a dictionary with student IDs as
5    # the keys and student names as the values.
6    students = {
7    "42-039-4736": "Clint Huish",
8    "61-315-0160": "Amelia Davis",
9    "10-450-1203": "Ana Soares",
10   "75-421-2310": "Abdul Ali",
11   "07-103-5621": "Amelia Davis"
12   }
13   # Attempt to find the key "50-420-1021",
14   # which is not in the dictionary. This will
15   # cause the computer to raise a KeyError.
16   name = students["50-420-1021"]
17   print(name)
18  except KeyError as key_err:
19   print(type(key_err).__name__, key_err)
20if __name__ == "__main__":
21  main()
```

```
> python key_error.py
KeyError '50-420-1021'
```

Of course, it is very unlikely that a programmer would write a program that tries to find a hard-coded key that is not in a dictionary. However, it is common for a user to enter a key that is not in a dictionary. This is why the programs in examples 1 and 4 in the prepare content for lesson 8 include an **if** statement above the line of code that searches the dictionary, like this:

```
# Get a student ID from the user.
id = input("Enter a student ID: ")
# Check if the student ID is in the dictionary.
if id in students:
  # Find the student ID in the dictionary and
  # retrieve the corresponding student name.
  name = students[id]
  # Print the student's name.
  print(name)
else:
  print("No such student")
```

FileNotFoundError

If we write a call to the **open** function that attempts to open a file for reading and that file doesn't exist, the computer will raise a **FileNotFoundError**. Example 8 contains code where such an error might occur.

```
# Example 8
def main():
  try:
    with open("products.vcs", "rt") as products_file:
    for row in products_file:
    print(row)
  except FileNotFoundError as not_found_err:
    print(not_found_err)
if __name__ == "__main__":
  main()
```

```
> python file_not_found.py
[Errno 2] No such file or directory: 'products.vcs'
```

PermissionError

Nearly all computer operating systems, such as Microsoft Windows, Mac OS X, and Linux, allow multiple people to use a single computer. Because people need to store private data in files on a computer, the operating systems implement file access permission rules. These rules help to prevent unauthorized access to files.

If we write a call to the **open** function that attempts to open a file and the person executing our program doesn't have permission to access the file, the computer will

raise a **PermissionError**. Example 9 contains code where such an error might occur.

```
# Example 9
def main():
  try:
    with open("contacts.csv", "rt") as contacts_file:
    for row in contacts_file:
    print(row)
  except PermissionError as perm_err:
    print(perm_err)
if __name__ == "__main__":
  main()
```

```
> python permission_error.py
[Errno 13] Permission denied: 'contacts.csv'
```

Example: Arithmetic

Example 10 contains a complete program with **except** blocks to handle two types of exceptions: **ValueError** and **ZeroDivisionError**.

```
# Example 10
"""
The owner of Sam's Sandwich Shop requested this program,
which computes the number of sandwiches per employee
that his employees made in his restaurant in one day.
"""
def main():
  try:
    # Get the number of sandwiches made today and the
    # number of employees who worked today from the user.
    sandwiches = int(input("Number of sandwiches made today: "))
    employees = int(input("Number of employees who worked today:
"))
    # Compute the number of sandwiches per employee
    # that were made today in the restaurant.
    sands_per_emp = sandwiches / employees
    # Print the results for the user to see.
    print(f"{sands_per_emp:.1f} sandwiches per employee")
  except ValueError as val_err:
    print(f"Error: {val_err}")
    print("You entered text that is not an integer. Please")
    print("run the program again and enter an integer.")
```

```
  except ZeroDivisionError as zero_div_err:
    print(f"Error: {zero_div_err}")
    print("You entered 0 for the number of employees.")
    print("Please run the program again and enter an integer")
    print("larger than 0 for the number of employees.")
    # Call main to start this program.
if __name__ == "__main__":
  main()
```

```
> python example_10.py
Number of sandwiches that were made today: 35u
Error: invalid literal for int() with base 10: '35u'
You entered text that is not an integer. Please
run the program again and enter an integer.
> python example_10.py
Number of sandwiches made today: 350.4
Error: invalid literal for int() with base 10: '350.4'
You entered text that is not an integer. Please
run the program again and enter an integer.
> python example_10.py
Number of sandwiches that were made today: 350
Number of employees who worked today: 0
Error: division by zero
You entered 0 for the number of employees.
Please run the program again and enter an integer
larger than 0 for the number of employees.
> python example_10.py
Number of sandwiches that were made today: 350
Number of employees who worked today: 8
43.8 sandwiches per employee
```

Example: Reading from a File

The program in example 11 below handles exceptions that might occur when the program opens and reads from a file. This program contains only one **`try`** block, which begins at line 8 and includes all the regular code in the **`main`** function. This one **`try`** block has three **`except`** blocks at lines 36, 38, and 40 that handle **`FileNotFoundError`**, **`PermissionError`**, and **`ZeroDivisionError`**.

```
# Example 11
import csv
DATE_INDEX = 0
START_MILES_INDEX = 1
END_MILES_INDEX = 2
GALLONS_INDEX = 3
```

```
def main():
    try:
        # Open the fuel_usage.csv file.
        filename = "fuel_usage.csv"
        with open(filename, "rt") as usage_file:
        # Use the standard csv module to get
        # a reader object for the CSV file.
        reader = csv.reader(usage_file)
        # The first line of the CSV file contains
        # headings and not fuel usage data, so this
        # statement skips the first line of the file.
        next(reader)
        # Print headers for the three columns.
        print("Date,Start,End,Gallons,Miles/Gallon")
        # Process each row in the CSV file.
        for row_list in reader:
        # From the current row of the CSV file, get
        # the date, the starting and ending odometer
        # readings, and the number of gallons used.
        date = row_list[DATE_INDEX]
        start_miles = float(row_list[START_MILES_INDEX])
        end_miles = float(row_list[END_MILES_INDEX])
        gallons = float(row_list[GALLONS_INDEX])
        # Call the miles_per_gallon function.
        mpg = miles_per_gallon(
        start_miles, end_miles, gallons)
        # Display the results for one row.
        mpg = round(mpg, 1)
        print(date, start_miles, end_miles, gallons, mpg, sep=",")
    except FileNotFoundError as not_found_err:
        print(f"Error: cannot open {filename} because it doesn't
exist.")
    except PermissionError as perm_err:
        print(f"Error: cannot read from {filename} because you don't
have permission.")
    except ZeroDivisionError as zero_div_err:
        print(f"Error: {filename} contains a zero in the gallons
column.")
def miles_per_gallon(start_miles, end_miles, gallons):
    """Compute and return the average number of miles
    that a vehicle traveled per gallon of fuel.
    Parameters
    start_miles: starting odometer reading in miles.
    end_miles: ending odometer reading in miles.
    gallons: amount of fuel used in U.S. gallons.
    Return: miles per gallon
    """
    mpg = abs(end_miles - start_miles) / gallons
```

```
52    return mpg
53# Call main to start this program.
54if __name__ == "__main__":
55    main()
```

Validating User Input

To *validate user input* means to check user input to ensure it is in the correct format before using that input. The program in example 12 validates user input by handling exceptions. Notice in the **`get_float`** function, there is a **`try`** block at line 23. The **`try`** block is part of a loop that validates user input in the **`get_float`** function. Notice at line 37 that the **`except`** block handles **`ValueError`** which is the type of exception that the **`float`** function raises when it tries to convert text to a number but the text contains characters that are not numeric.

```
 1# Example 12
 2def main():
 3    gender = input("Enter your gender (M or F): ")
 4    weight = get_float("Enter your weight in kg: ", 20, 500)
 5    height = get_float("Enter your height in cm: ", 60, 250)
 6    age = get_float("Enter your age in years: ", 10, 120)
 7    bmr = basal_metabolic_rate(gender, weight, height, age)
 8    print(f"Your basal metabolic rate is {bmr} calories per day.")
 9def get_float(prompt, lower_bound, upper_bound):
10    """Get a number from the user, validate that the user
11    entered a number and not some other text, validate that
12    the number is between a lower and upper bound, and then
13    return the number. If the user enters an invalid number,
14    this function will prompt the user repeatedly until the
15    user enters a valid number.
16    Parameters
17    prompt: A string to display to the user.
18    lower_bound: The smallest number that the user may enter.
19    upper_bound: The largest number that the user may enter.
20    Return: The number entered by the user.
21    """
22    while True:
23        try:
24            text = input(prompt)
25            number = float(text)
26            if number < lower_bound:
27                print(f"{number} is too small.")
28                print("Please enter another number.")
29            elif number > upper_bound:
```

```
        print(f"{number} is too large.")
        print("Please enter another number.")
      else:
        # If the computer gets to this line of code,
        # the user entered a valid number between
        # lower_bound and upper_bound, so exit the loop.
        break
    except ValueError as val_err:
      # The user entered at least one character that is
      # not part of a floating point number, so print a
      # message asking the user to enter a number.
      print(f"{text} is not a number.")
      print("Please enter a number.")
  return number
def basal_metabolic_rate(gender, weight, height, age):
  """Calculate and return a person's basal metabolic rate
  in calories per day. weight must be in kilograms, height
  must be in centimeters, and age must be in years.
  """
  if gender.upper() == "F":
    bmr = 447.593 + 9.247 * weight \ + 3.098 * height - 4.330 *
age
  else:
    bmr = 88.362 + 13.397 * weight \   + 4.799 * height - 5.677 *
age
  return bmr
# Call main to start this program.
if __name__ == "__main__":
  main()
```

Tutorials

If the concepts above seem vague, these tutorials may clear some confusion for you:

- Python Exceptions: An Introduction
- Official Python Errors and Exceptions tutorial
- The Most Diabolical Python Antipattern
- Understanding the Python Traceback

The official Python built-in exceptions reference contains a list of all the built-in exceptions. It also includes the class hierarchy for the built-in exceptions that is helpful for ordering **`except`** blocks from most specific to most general.

Summary

Errors and exceptional situations sometimes occur while a program is running. When an exceptional situation occurs, a computer will raise an exception. With the **`try`** and **`except`** keywords, you can write Python code that will handle exceptions. Write normal program code inside a **`try`** block and write an **`except`** block for each type of exception that you want your program to handle.

There are many types of exceptions in Python, but there are only seven types that your code will need to handle in CSE 111, namely:

- **`TypeError`**
- **`ValueError`**
- **`ZeroDivisionError`**
- **`IndexError`**
- **`KeyError`**
- **`FileNotFoundError`**
- **`PermissionError`**

When writing code that writes to or reads from a file, a programmer usually writes **`except`** blocks to handle **`FileNotFoundError`** and **`PermissionError`**. Also, when writing code that gets input from a user, a programmer usually writes **`try`** and **`except`** blocks to help validate the user's input.

Checkpoint

Purpose

Improve your understanding of how to handle exceptions in a Python program.

Assignment

Do the following:

1. Download and save the **`accidents.csv`** and **`get_line.py`** files in the same folder. **`get_line.py`** is a simple program that asks a user to input the name of a text file and a line number. Then it prints the text that is in the file on the requested line.
2. Open **`get_line.py`** in VS Code and notice the **`try`** block at line 10 and the five **`except`** blocks at lines 27–74, each handling a different type of exception.

3. Run the **`get_line.py`** program five times and enter the input shown in the Sample Run section below. For each of the first four times that you run the program, find the lines of code in **`get_line.py`** that handled the exception that was raised.

Sample Run

```
> python get_line.py
Enter the name of text file: notafile.csv
FileNotFoundError: [Errno 2] No such file or directory: 'notafile.csv'
The file notafile.csv doesn't exist.
Run the program again and enter the name of an existing file.
> python get_line.py
Enter the name of text file: accidents.csv
Enter a line number: hey
ValueError: invalid literal for int() with base 10: 'hey'
You entered an invalid integer for the line number.
Run the program again and enter an integer for the line number.
> python get_line.py
Enter the name of text file: accidents.csv
Enter a line number: -300
IndexError: list index out of range
-300 is a negative integer.
Run the program again and enter a line number between 1 and 7.
> python get_line.py
Enter the name of text file: accidents.csv
Enter a line number: 75
IndexError: list index out of range
75 is greater than the number of lines in accidents.csv.
There are only 7 lines in accidents.csv.
Run the program again and enter a line number between 1 and 7.
> python get_line.py
Enter the name of text file: accidents.csv
Enter a line number: 4
2012,33782,2362000,5615000,31006,3167,440,9420,6396,1262
```

Submission

When complete, report your progress in the associated Canvas quiz.

Useful Links:

- Return to: Week Overview | Course Home