10 Prepare: Handling 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.

Error Handling Concepts (13 minutes)

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 Python program is running. For example, an exception occurs when a Python program tries to open a file for reading, and 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.

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
 2
 3  try:
 4      # Write normal code here. This block must include
 5      # code that falls into two groups:
 6      # 1. Code that may cause an exception to be raised
 7      # 2. Code that depends on the results from the code
 8      #    in the first group
 9  except ZeroDivisionError as zero_div_err:
10      # Code that the computer executes if the code in the
11      # try block caused a ZeroDivisionError to be raised.
12  except ValueError as val_err:
13      # Code that the computer executes if the code in
14      # the try block caused a ValueError to be raised.
15  except (TypeError, KeyError, IndexError) as error:
16      # Code that the computer executes if the code in the try
```

```
17      # block raised a TypeError, KeyError, or IndexError.
18  except Exception as excep:
19      # Code that the computer executes if the code in the
20      # try block caused any exception to be raised that was
21      # not handled by one of the previous except blocks.
22  except:
23      # Code that the computer executes if the code in the
24      # try block caused anything to be raised that was
25      # not handled by one of the previous except blocks.
26  else:
27      # Code that the computer executes after the code
28      # in the try block if the code in the try block
29      # did not raise any exceptions.
30  finally:
31      # Code that is executed after all the other code in
32      # try, except, and else blocks regardless of whether
33      # 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 12:

```
except ValueError as val_err:
```

Or each `except` block may handle several types of exceptions, like the code at line 15:

```
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. However, in most programs, it is a bad idea to write `except` blocks that are very general, including an `except` block that handles all possible exception types and a bare `except` block.

It is usually a bad idea to write an `except` block that handles all types of exceptions or a bare except block because such a block will handle `SyntaxError`. Normally, a program should not handle `SyntaxError`. Instead, a program should crash for a syntax error and print the line number where the syntax error occurred so that a programmer can find and fix the syntax error. As explained in the prepare content for lesson 6, syntax errors are caused by a programmer mistyping code and not by bad user input or missing files. A programmer should find and fix all syntax errors in a program long before the program is given to users, so there is no reason to handle syntax errors in an `except` block.

As shown at line 26 in example 1 above, following the `except` blocks, we may write an optional `else` block which the computer will execute if the `try` block does not raise any exceptions. It is uncommon to need to write code in the `else` block of `try` and `except`, and professional programmers almost never do it.

As shown at line 30 in example 1 above, at the end of the exception handling code, we 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.

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 type. The code in example 2 attempts to pass a string of text to the `round` function which causes the computer to raise `TypeError` as shown in the output below the code example.

```
# 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
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 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

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__":
    main()
```

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

KeyError

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

```
# Example 7

def main():
    try:
        # Create a dictionary with student IDs as
        # the keys and student names as the values.
        students = {
            "42-039-4736": "Clint Huish",
            "61-315-0160": "Michelle Davis",
            "10-450-1203": "Jorge Soares",
            "75-421-2310": "Abdul Ali",
            "07-103-5621": "Michelle Davis"
        }

        # Attempt to find the key "50-420-1021",
        # which is not in the dictionary. This will
        # cause the computer to raise a KeyError.
        name = students["50-420-1021"]

        print(name)

    except KeyError as key_err:
        print(type(key_err).__name__, key_err)

if __name__ == "__main__":
    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 code in examples 1 and 4 in the prepare content for lesson 8 includes 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 line in products_file:
                print(line)

    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 line in contacts_file:
                print(line)

    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
"""
Sam, who is the owner of Sam's Sandwich Shop, requested
this program, which computes the number of sandwiches per
employee that were 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 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 12 and includes all the regular code in the `main` function. This one `try` block has three `except` blocks at lines 48, 52, and 56 that handle `FileNotFoundError`, `PermissionError`, and `ZeroDivisionError`.

```
 1  # Example 11
 2
 3  import csv
 4
 5  DATE_INDEX = 0
 6  START_MILES_INDEX = 1
 7  END_MILES_INDEX = 2
 8  GALLONS_INDEX = 3
 9
10
11  def main():
12      try:
13          # Open the fuel_usage.csv file.
```

```
14          filename = "fuel_usage.csv"
15          with open(filename, "rt") as usage_file:
16
17              # Use the standard csv module to get
18              # a reader object for the CSV file.
19              reader = csv.reader(usage_file)
20
21              # The first line of the CSV file contains headings
22              # and not fuel usage data, so this statement skips
23              # the first line of the CSV file.
24              next(reader)
25
26              # Print headers for the three columns.
27              print("Date,Start,End,Gallons,Miles/Gallon")
28
29              # Process each row in the CSV file.
30              for row in reader:
31
32                  # From the current row of the CSV file, get
33                  # the date, the starting and ending odometer
34                  # readings, and the number of gallons used.
35                  date = row[DATE_INDEX]
36                  start_miles = float(row[START_MILES_INDEX])
37                  end_miles = float(row[END_MILES_INDEX])
38                  gallons = float(row[GALLONS_INDEX])
39
40                  # Call the miles_per_gallon function.
41                  mpg = miles_per_gallon(start_miles, end_miles, gallons)
42
43                  # Display the results for one row.
44                  mpg = round(mpg, 1)
45                  print(date, start_miles, end_miles, gallons, mpg,
46                          sep=",")
47
48      except FileNotFoundError as not_found_err:
49          print(f"Error: cannot open {filename}"
50                  " because it doesn't exist.")
51
52      except PermissionError as perm_err:
53          print(f"Error: cannot read from {filename}"
54                  " because you don't have permission.")
55
56      except ZeroDivisionError as zero_div_err:
57          print(f"Error: {filename} contains a"
58                  " zero in the gallons column.")
59
60
61  def miles_per_gallon(start_miles, end_miles, gallons):
62      """Compute and return the average number of miles
63      that a vehicle traveled per gallon of fuel.
64      start_miles and end_miles are odometer readings in miles.
65      gallons is a fuel amount in U.S. gallons.
66      """
67      mpg = abs(end_miles - start_miles) / gallons
68      return mpg
69
70
71  # Call main to start this program.
72  if __name__ == "__main__":
73      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 at line 28 there is a `try` block. The `try` block is part of a loop that validates user input in the `get_float` function. Notice at line 43 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.

```python
# Example 12

def main():
    gender = input("Enter your gender (M or F): ")

    weight = get_float("Enter your weight in kilograms: ", 20, 500)
    height = get_float("Enter your height in centimeters: ", 60, 250)
    age = get_float("Enter your age in years: ", 10, 120)

    bmr = basal_metabolic_rate(gender, weight, height, age)
    print(f"Your basal metabolic rate is {bmr} calories per day.")


def get_float(prompt, lower_bound, upper_bound):
    """Get a number from the user, validate that the user entered
    a number and not some other text, validate that the number is
    between a lower and upper bound, and then return the number. If
    the user enters an invalid number, this function will prompt
    the user repeatedly until the user enters a valid number.

    Parameters
        prompt: A string to display to the user.
        lower_bound: The smallest number that the user may enter.
        upper_bound: The largest number that the user may enter.
    Return: The number entered by the user.
    """
    while True:
        try:
            text = input(prompt)
            number = float(text)
            if number < lower_bound:
                print(f"{number} is too small.")
                print("Please enter another number.")
            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 (bmr)
    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. Put 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: `TypeError`, `ValueError`, `ZeroDivisionError`, `IndexError`, `KeyError`, `FileNotFoundError`, and `PermissionError`. When writing code that gets input from a user, a programmer usually writes `try` and `except` blocks to help validate the user's input. Also, when writing code that writes to or reads from a file, a programmer usally writes `try` and `except` blocks to handle `FileNotFoundError` and `PermissionError`.