Infrastructure as Code- Walkthrough

Handling Errors

Contents

Last Updated: 6 October 2022

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started coding in Python.

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed VSC as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
    - **a.** This could be on your OneDrive, in these examples, I am using **OneDrive\Python\Exercises_07**

Writing Utilities

I'm going to take some code from Exercise_06 and I'm going to start building up some OS utilities. Save the new file under **Exercises_07** as **detect_working_directory.py**

Consider the changes.

1. I've added an import to the **sys** and **os** libraries.
2. I'm going to define global variables which I can use anywhere in the code.
3. I am going to exit the programme unless I detect Windows or Linux.
4. I have added a simple function to get my working path and print it for diagnostics.

```
"""
directory_utilities.py
By: JOR
Date: 01OCT22
"""

import os, platform

# Define global variables
current_working_directory = None

def detect_os()->str:
    # Detect the OS in use
    return platform.system()

def detect_working_directory()->str:
    # Returns the directory this script was run from
    return os.getcwd()

if (__name__ == '__main__'):
    print(f"This module executes as a standalone script")

    # Check the OS in use, lower case
    my_os = detect_os()
    my_os = my_os.lower()

    # Parse the response, only check for Windows and Linux
    if my_os == "windows":
        print("Your system is Windows")
    elif my_os == "linux":
        print("Your system is Linux")
    else:
        print(f"Cannot continue, unidentified system = {my_os}")
        sys.exit()

    # Get the current working directory
    current_working_directory = detect_working_directory()
    print(f"You are coding in: {current_working_directory}")

else:
    print(f"This module is called {__name__} and is being called by another script")
```

So far, I'm just taking responses from the operating system. the **if** statement is now only checking for Linux and Windows. The **else** option deals with any other OS and exits the program.

Systematically Handling Errors

If code has been written incorrectly, the interpreter will catch this and label it as a syntax error. Even if your Python is syntactically correct, it may still generate errors when you try to run it. Perhaps you are running in the wrong operating system, or missing libraries? An exception is a run time error which occurs during the execution of the program. In my code, I handle such events in a planned way.

There are keywords used for exception handling:

Try is the block of code we are about to attempt to run.

Except is the code that runs if an error occurs and should handle the error gracefully. For example, it may report the error in detail and then resume the programme. You can have a separate Except section for each error type.

Else is the additional code that runs if an error does not occur.

Finally is the code that runs at the end, whether there is an error or not.

I am going to write a simple function to create a directory but I'm going to handle any errors in its creation. Copy the content of **detect_working_directory.py** to a new file **create_directoy.py**

Add the following function.

```
def create_directory(directory_name: str) ->bool:
    # Use try/except to catch errors
    try:
        # Create the diretory
        os.mkdir(directory_name)
        # If this worked, return True
        return True
    except:
        # Give an explicit error message
        print("Error creating directory {directory_name}")
        # If this did not work, return False
        return False
```

At the end of your **if (__name__ == '__main__'):** section, with the correct indentation, add

```
if create_directory("JOR"):
        print("Creating a directory worked")
        # Do other stuff
    else:
        print("You couldn't create a directory!")
        # Do other stuff
```

When you run the program the first time, it works. However, the second time you run the program, it fails. You already created the directory; you cannot create a directory with the same name in the same path.

The “already existing” case wasn’t really an error, we should deal with this case differently. I can modify my function to return a value if the directory already exists. Maybe I’m going to return an integer and change my calling code!

```
def create_directory(directory_name: str)->int:
    # Check to see if the directory already exists
    if os.path.isdir(directory_name):
        # The directory already exists
        return 2
    else:
        # Use try/except to catch errors
        try:
            # Create the directory
            os.mkdir(directory_name)
            # If this worked, return True
            return 0
        except:
            # Give an explicit error message
            print("Error creating directory {directory_name}")
            # If this did not work, return False
            return 1
```

There are now three possible return statuses I could have.

```
if create_directory("JOR") == 0:
        print("Creating a directory worked")
        # Do other stuff
    elif create_directory("JOR") == 1:
        print("You couldn't create a directory!")
        # Do other stuff
    elif create_directory("JOR") == 2:
        print("Directory already existed!")
        # Do other stuff
```

Working with files

One of the things we must do in many programs, is to open a file for reading or writing.

There are four modes in which you can open a file:

- Read = “r”
- Write = “w”
- Append = “a”
- Read and write = “rw”

```
my_filename = "logfile.txt"

try:
    with open(my_filename, "a") as file_handle:
        print(f"Writing a test line to {my_filename}")
        file_handle.write("Test line")
except IOError as err:
    print(f"IOError was {err}")
except EOFError as err:
    print(f"End of file error was {err}")
except OSError:
    print("OS Error")
except:
    print("General Error")
else:
    print("File created")
finally:
    print("Finishing up!")
    # close not needed because with statement
    # file_handle.close()
```

Notice the use of the **with** statement, this has special characteristics for use with streaming IO, files, serial ports etc.

Try this code out, it should create a log file and give you messages for each section which runs. When you open a file, you can open to write (”w”) or read (“r”).

In the open command, change the “w” to an “r”, you are now opening the file read only.

When you attempt to write to the file, this will generate an operating system error. But more specifically, it generates an IO Error. In this example, I take the error message back and provide it to the user.

In this example I specifically handle some common file related errors.

Any other error is handled as a general error using the **except** keyword.

Review the examples at https://docs.python.org/3/tutorial/errors.html#

Input Validation

One of the ways we intercept errors is to validate user input.

```
def validate_integer():
    # Loop forever
    while True:
        try:
            user_input = int(input("Enter an integer value: "))
        except:
            # Bad value,
            print("Error")
            continue
        else:
            print("Valid input")
            # Good value, exit the loop
            break
        finally:
            # Only runs after the except, continue
            print("Try again - enter an integer value only!")


validate_integer()
```

Run this code. Try entering the value **10**, then run the code again and enter the value **ten**.

Exercise

I have two variables on my diesel backup generator.

- **fuel** in litres
- **fuel_consumption** in litres per minute.

I can calculate my remaining endurance in minutes as

$$Endurance = \frac{Fuel}{Fuel\ Consumption}$$

Whenever the motor is idling, the flowmeter cannot calculate fuel flow and sets it = 0.

Write a function to calculate the remaining endurance in minutes, checking and handling for divide by zero errors and for any value errors.

Raising Exceptions

You can also raise an exception when validating input.

```
# Take an input number as a string and convert it to an integer
my_value = int(input("Enter an integer greater than 0"))

if my_value <= 0:
    raise Exception("Values must be > 0")
else:
    print("Validation checks passed")
```