Assignment No. 11

Submitted By: Sachin Dodake

1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.

`Ans-1`

> - The purpose of the `else block` in `try-except` is to execute code that should only run if no exceptions were raised in the try block.
> - The `else block` is optional, and it follows all the except blocks in a `try-except` statement.
> - If `no exceptions` are raised in the try block, the code in the `else block is executed`. This is useful for cases where you want to perform some action only if the try block completes successfully.
> - **Example:**

In [1]:
```python
try:
    print(f"Case-1")
    num1 = int(input("\tEnter the first number: "))
    num2 = int(input("\tEnter the second number: "))
    result = num1 / num2

except ZeroDivisionError:
    print("\tError: You cannot divide by zero.")

else:
    print("\tThe result is:", result)

######################################################

try:
    print(f"Case-2")
    num1 = int(input("\tEnter the first number: "))
    num2 = int(input("\tEnter the second number: "))
    result = num1 / num2

except ZeroDivisionError:
    print("\tError: You cannot divide by zero.")

else:
    print("\tThe result is:", result)
```

```
Case-1
	Enter the first number: 22
	Enter the second number: 2
	The result is: 11.0
Case-2
	Enter the first number: 22
	Enter the second number: 0
	Error: You cannot divide by zero.
```

2. Can a try-except block be nested inside another try-except block? Explain with an example.

`Ans-2`

> - Yes, we can nested a try-except block within another try-except block, this is called nested try-except block.
> - In this case, if an exception is raised in the nested try block, the nested except block is used to handle it. In case the nested except is not able to handle it, the outer except blocks are used to handle the exception.
> - **Example:**

In [9]:
```python
x = 10
y = 0

try:
    print("OUTER TRY BLOCK")
    try:
        print("\tINNER TRY BLOCK")
        print(x / y)
    except TypeError as te:
        print("\tINNER EXCEPT BLOCK")
        print(te)
except ZeroDivisionError as ze:
    print("OUTER EXCEPT BLOCK")
    print(ze)
```

```
OUTER TRY BLOCK
	INNER TRY BLOCK
OUTER EXCEPT BLOCK
division by zero
```

3. How can you create a custom exception class in Python? Provide an example that demonstrates its usage.

`Ans-3`

> `Custom Exceptions:`
>
> - In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

> **Example with demonstration:**
>
> - If the user input input_num is greater than 18, then there will be NO exception and program runs as expected.
> - If the user input input_num is smaller than 18, then it will raise an exception.
> - In example below, we have defined the custom exception `InvalidAgeException` by creating a new class that is derived from the built-in `Exception` class.
> - Here, when `input_num` is smaller than `18`, this code generates an exception.
> - When an exception occurs, the rest of the code inside the `try block` is skipped.
> - The `except block` catches the user-defined `InvalidAgeException` exception and statements inside the `except block` are executed.

In [13]:
```python
# define Python user-defined exceptions
class InvalidAgeException(Exception):
    "Raised when the input value is less than 18"
    pass

# you need to guess this number
number = 18

try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")

except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

```
Enter a number: 17
Exception occurred: Invalid Age
```

4. What are some common exceptions that are built-in to Python?

`Ans-4`

> `Exceptions:`
>
> - Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.
> - Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur.
> - We can view all the built-in exceptions using the built-in local() function as follows:

```
print(dir(locals()['__builtins__']))
```

> - Here, locals()['**builtins**'] will return a module of built-in exceptions, functions, and attributes and dir allows us to list these attributes as strings.
> - The various exceptions and their causes are listed below:

> - `AssertionError:` Raised when an assert statement fails.
> - `AttributeError:` Raised when attribute assignment or reference fails.
> - `EOFError:` Raised when the input() function hits end-of-file condition.
> - `FloatingPointError:` Raised when a floating point operation fails.
> - `GeneratorExit:` Raise when a generator's close() method is called.
> - `ImportError:` Raised when the imported module is not found.
> - `IndexError:` Raised when the index of a sequence is out of range.
> - `KeyError:` Raised when a key is not found in a dictionary.
> - `KeyboardInterrupt:` Raised when the user hits the interrupt key (Ctrl+C or Delete).
> - `MemoryError:` Raised when an operation runs out of memory.
> - `NameError` Raised when a variable is not found in local or global scope.
> - `NotImplementedError` Raised by abstract methods.
> - `OSError` Raised when system operation causes system related error.
> - `OverflowError` Raised when the result of an arithmetic operation is too large to be represented.
> - `ReferenceError` Raised when a weak reference proxy is used to access a garbage collected referent.
> - `RuntimeError` Raised when an error does not fall under any other category.
> - `StopIteration` Raised by next() function to indicate that there is no further item to be returned by iterator.
> - `SyntaxError` Raised by parser when syntax error is encountered.
> - `IndentationError` Raised when there is incorrect indentation.
> - `TabError` Raised when indentation consists of inconsistent tabs and spaces.
> - `SystemError` Raised when interpreter detects internal error.
> - `SystemExit` Raised by sys.exit() function.
> - `TypeError` Raised when a function or operation is applied to an object of incorrect type.
> - `UnboundLocalError` Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
> - `UnicodeError` Raised when a Unicode-related encoding or decoding error occurs.
> - `UnicodeEncodeError` Raised when a Unicode-related error occurs during encoding.
> - `UnicodeDecodeError` Raised when a Unicode-related error occurs during decoding.
> - `UnicodeTranslateError` Raised when a Unicode-related error occurs during translating.
> - `ValueError` Raised when a function gets an argument of correct type but improper value.
> - `ZeroDivisionError` Raised when the second operand of division or modulo operation is zero.

5. What is logging in Python, and why is it important in software development?

`Ans-5`

> `Logging:`
>
> - Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging, and running. If you don't have any logging record and your program crashes, there are very few chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging, you can leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.

> `Levels of Log Message:`
>
> - `Debug (Level-10):` These are used to give Detailed information, typically of interest only when diagnosing problems.
> - `Info (Level-20) :` These are used to confirm that things are working as expected.
> - `Warning (Level-30) :` These are used an indication that something unexpected happened, or is indicative of some problem in the near future.
> - `Error (Level-40) :` This tells that due to a more serious problem, the software has not been able to perform some function.
> - `Critical (Level-50) :` This tells serious error, indicating that the program itself may be unable to continue running.

> `Importance:`
>
> - Logging is essential to understand the behaviour of the application and to debug unexpected issues or for simply tracking events. In the production environment, we can't debug issues without proper log files as they become the only source of information to debug some intermittent or unexpected errors.
> - Helps Troubleshooting Bugs.
> - Improves Monitoring Projects in Production Environments.
> - Facilitates Debugging.
> - You can easily control the format of the messages you log. The module comes with a lot of useful attributes that you can decide to include in your log.
> - You can log messages with different levels of urgency/warning/information. This allows you to categorize log messages easily and makes your debugging life way easier.
> - You can set the destination of your logs to pretty much anything. Even to sockets.
> - Doesn't tamper with the user experience if your module is being imported by other users.

6. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.

`Ans-6`

> - There are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

```
1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL
```

> - **Example:** The basic example on logging is as below: While there are five lines of logging, you may see only three lines of output if you run this script. This is because the root logger, by default, only prints the log messages of a severity level of WARNING or above. However, using the root logger this way is not much different from using the `print()` function.

In [1]:
```python
import logging

logging.debug('Debug message')
logging.info('Info message')
logging.warning('Warning message')
logging.error('Error message')
logging.critical('Critical message')
```

```
WARNING:root:Warning message
ERROR:root:Error message
CRITICAL:root:Critical message
```

7. What are log formatters in Python logging, and how can you customise the log message format using formatters?

`Ans-7`

> `Log Formatters:`
>
> - The formatters are used to define the structure of the output. It is used the string formatting methods to specify the format of the log messages.
> - To configure the format of the logger, we use a Formatter. It allows us to set the format of the log, similarly to how we did so in the root logger's basicConfig().
> - First, we create a formatter, then set our handler to use that formatter. If we wanted to, we could make several different loggers, handlers, and formatters so that we could mix and match based on our preferences.

In [4]:
```python
import logging

# Set up root logger, and add a file handler to root logger
logging.basicConfig(filename = 'file.log',
                    level = logging.WARNING,
                    format = '%(asctime)s:%(levelname)s:%(name)s:%(message)s')

# Create logger, set level, and add stream handler
parent_logger = logging.getLogger("parent")
parent_logger.setLevel(logging.INFO)
parent_fhandler = logging.FileHandler('parent.log')
parent_fhandler.setLevel(logging.WARNING)
parent_formatter = logging.Formatter('%(asctime)s:%(levelname)s:%(message)s')
parent_fhandler.setFormatter(parent_formatter)
parent_logger.addHandler(parent_fhandler)

# Log message of severity INFO or above will be handled
parent_logger.debug('Debug message')
parent_logger.info('Info message')
parent_logger.warning('Warning message')
parent_logger.error('Error message')
parent_logger.critical('Critical message')
```

> - After excecution of the above code, we will get two log files i.e. `file.log` and `parent.log` the output as below:

> - `file.log`

```
2023-06-24 11:36:24,546:INFO:parent:Info message
2023-06-24 11:36:24,546:WARNING:parent:Warning message
2023-06-24 11:36:24,546:ERROR:parent:Error message
2023-06-24 11:36:24,547:CRITICAL:parent:Critical message
```

> - `parent.log`

```
2023-06-24 11:36:24,546:WARNING:Warning message
2023-06-24 11:36:24,546:ERROR:Error message
2023-06-24 11:36:24,547:CRITICAL:Critical message
```

8. How can you set up logging to capture log messages from multiple modules or classes in a Python application?

`Ans-8`

> `Log Messages from Multiple Modules:`
>
> - A well-organized Python application is likely composed of many modules. Sometimes these modules are intended to be used by other programs. Still, unless you're intentionally developing reusable modules inside your application, you're likely using modules available from PyPI and modules that you write yourself specifically for your application.
> - In general, a module should emit log messages as a best practice and should not configure how those messages are handled. That is the responsibility of the application.
> - The module needs two lines to set up logging, and then use the named logger:

```
import logging
log = logging.getLogger(__name__)
def do_something():
        log.debug("Doing something!")
```

> `Setup:`
>
> - To set up basic logging onto a file, you can use the basicConfig() constructor, as shown below.

In [2]:
```python
import logging

logging.basicConfig(filename="py_log.log",
                    level=logging.INFO,
                    filemode="w",
                    format="%(asctime)s %(levelname)s %(message)s")
logging.debug("A DEBUG Message")
logging.info("An INFO")
logging.warning("A WARNING")
logging.error("An ERROR")
logging.critical("A message of CRITICAL severity")
```

> - Here, the purpose of various parameters are as below:

```
1. **`level:`** This is the level you'd like to start logging at. If this is set to info,
then all messages corresponding to debug are ignored.
2. **`filename:`** The parameter filename denotes the file handler object. You can specify
the name of the file to log onto.
3. **`filemode:`** This is an optional parameter specifying the mode in which you'd like
to work with the log file specified by the parameter filename. Setting the file mode to it
write (w) will overwrite the logs every time the module is run. The default filemode is
append (a) which means you'll have a log record of events across all runs of the program.
```

> - After running the main module, you'll see that the log file py_log.log has been created in the current working directory.
> - Since we set the logging level to info, the log record now contains the message corresponding to INFO as below:

```
2023-06-24 11:52:09,571 INFO An INFO
2023-06-24 11:52:09,571 WARNING A WARNING
2023-06-24 11:52:09,571 ERROR An ERROR
2023-06-24 11:52:09,571 CRITICAL A message of CRITICAL severity
```

> - The logs in the log file are of the format: **<timestamp>:<logging-level>:<name-of-the-logger>:<message>**. The **<name-of-the-logger>** is by default the root logger, as we haven't yet configured custom loggers.

9. What is the `difference` between the `logging` and `print` statements in Python? When should you use logging over print statements in a real-world application?

`Ans-9`

> `Logging in Python`:
>
> - Record events and errors that occur during the execution of Python programs.
> - Mainly used in the production environment.
> - Some features are: Log levels, filtering, formatting, and more.
> - It provides different log levels such as Debug, Info, Error, Warning, and Critical.
> - **Example:**

In [8]:
```python
import logging;
logging.basicConfig(level=logging.INFO);
logging.info("Hello")

# Output:

# Can be configured to log to different output destinations (e.g. console, file, network)
```

> `Print in Python`:
>
> - Displays the information to the console for the debugging purposes.
> - Mainly for debugging.
> - There are no good features.
> - It does not have any levels, it simply prints whatever is passed to it.
> - **Example:**

In [10]:
```python
print("Hello World!")

# Output:

# Prints only on the console
```

```
Hello World!
```

10. Write a Python program that `logs a message` to a file named `"app.log"` with the following requirements:

```
• The log message should be "Hello, World!"
• The log level should be set to "INFO."
• The log file should append new log entries without overwriting previous ones.
```

`Ans-10`

> - The program that record a log message "Hello World!" is as below:

In [3]:
```python
import logging

logging.basicConfig(filename="app.log",                          # setting filename as `app.log`
                    level=logging.INFO,                                          # seting level as `INFO`
                    filemode="w",                                                    # setting filemode a
                    format="%(asctime)s %(levelname)s %(message)s")  # setting th format of log lines
logging.debug("A DEBUG Message")
logging.info("Hello World!")
logging.warning("A WARNING")
logging.error("An ERROR")
logging.critical("A message of CRITICAL severity")
```

> - The output of this program is as below:

```
        2023-06-24 13:51:52,516 INFO Hello World!
# required msg
        2023-06-24 13:51:52,517 WARNING A WARNING
        2023-06-24 13:51:52,517 ERROR An ERROR
        2023-06-24 13:51:52,517 CRITICAL A message of CRITICAL severity
        2023-06-24 13:51:57,682 INFO Hello World!
# loading log file 2nd time
        2023-06-24 13:51:57,682 WARNING A WARNING
        2023-06-24 13:51:57,684 ERROR An ERROR
        2023-06-24 13:51:57,684 CRITICAL A message of CRITICAL severity
        2023-06-24 13:52:01,580 INFO Hello World!
# loading log file 3rd time
        2023-06-24 13:52:01,581 WARNING A WARNING
        2023-06-24 13:52:01,581 ERROR An ERROR
        2023-06-24 13:52:01,581 CRITICAL A message of CRITICAL severity
```

11. Create a Python program that logs an error message to the console and a file named "errors.log" if an exception occurs during the program's execution. The error message should include the exception type and a timestamp

In [1]:
```
import logging

# Configure logging
logging.basicConfig(level=logging.ERROR,
                    filename='errors.log',
                    filemode='a',
                    format='%(asctime)s-%(levelname)s-%(name)s-%(message)s')

try:
    num_1 = int(input("Enter First Number: "))
    num_2 = int(input("Enter Second Number: "))
    result = num_1/num_2
    raise ZeroDivisionError("Idiot, we can't divide a number by Zero!")

except Exception as e:
    error_msg = "{}-{}".format(e.__name__ , str(e))
    logging.error(str(error_msg), exc_info=True)
    print(f"Error Message: {error_msg}")
```

```
Enter First Number: 99
Enter Second Number: 0
Error Message: ZeroDivisionError - division by zero
```