Assignment 7

Q1. What is the purpose of the try statement?

Answer: The try statement catches and recovers from exceptions—it specifies a block of code to run, and one or more handlers for exceptions that may be raised during the block's execution.

When a try statement is entered, Python marks the current program context so it can return to it if an exception occurs. After the statements under the try block runs:

- If an exception does occur while the try block's statements are running, Python jumps back to the try and runs the statements under the first except clause that matches the raised exception. Control resumes below the entire try statement after the except block runs (unless the except block raises another exception).
- If an exception happens in the try block and no except clause matches, the exception is propagated up to the last matching try statement that was entered in the program or, if it's the first such statement, to the top level of the process (in which case Python terminates the program and prints a default error message).
- If no exception occurs while the statements under the try header run, Python runs the statements under the else line (if present), and control then resumes below the entire try statement.

Q2. What are the two most popular try statement variations?

Answer: The two most popular try statements variations are:

- try/except/else:
    - The block under the **try** header represents the main action of the statement—the code we're trying to run.
    - The **except** clauses define handlers for exceptions raised during the try block, except clauses are

focused exception handlers, they catch exceptions that occur only within the statements in the associated try block. However, as the try block's statements can call functions coded elsewhere in a program, the source of an exception may be outside the try statement itself.

    - The **else** clause (if coded) provides a handler to be run if no exceptions occur. It also lets us know whether the flow of control has proceeded past a try statement because no exception was raised, or because an exception occurred and was handled.

- try/finally:
    - If a **finally** clause is included in a try, Python will always run its block of statements "on the way out" of the try statement, whether an exception occurred while the try block was running or not.
    - The try/finally form is useful when we want to be completely sure that an action will happen after some code runs, regardless of the exception behaviour of the program. In practice, it allows you to specify cleanup actions that always must occur, such as file closes and server disconnects.

Q3. What is the purpose of the raise statement?

Answer: The purpose of raise statement is to trigger exceptions explicitly, raise statements can be coded as shown below.

- raise <instance> : # Raise instance of class
- raise <class> : # Make and raise instance of class
- raise: Running a raise this way **reraises the current exception** and propagates it to a higher handler (or the default handler at the top, which stops the program with a standard error message).

NOTE: Regardless of how it's named, exceptions are always identified by instance objects, and at most one is active at any

given time. Once caught by an except clause anywhere in the program, an exception dies (i.e., won't propagate to another try), unless it's reraised by another raise statement or error.

Q4. What does the assert statement do, and what other statement is it like?

Answer: Assertions are typically used to verify program conditions during development. When displayed, their error message text automatically includes source code line information and the value listed in the assert statement.

NOTE: **assert** is mostly intended for trapping user-defined constraints, not for catching genuine programming errors, and can be thought of as a special case of raise statement.

Q5. What is the purpose of the with/as argument, and what other statement is it like?

Answer:

- The purpose of **with/as** argument is to specify the termination time or "cleanup" activities that must run regardless of whether an exception occurs in a processing step.
- The **with/as** argument is designed to be an alternative to a common **try/finally** statement.
- The with statement supports a richer object-based protocol for specifying both entry and exit actions around a block of code.

```
with expression [as variable]:
    <with-block>
```

- The *expression* here is assumed to return an object that supports the context manage- ment protocol. This object may also return a value that will be assigned to the name *variable* if the optional *as* clause is present.

- For example, file objects have a context manager that automatically closes the file after the with block regardless of whether an exception is raised:

```
myfile = open(r'C:\filename.txt')
try:
     for line  in myfile:
          print(line)
          ...more code  here...
finally:
     myfile.close()
```

The above code can be written using with/as statements as follows:

```
with  open(r'C:\filename.txt') as  myfile:
     for line in myfile:
          print(line)
          ...
```

Although file objects are automatically closed on garbage collection, it’s not always straightforward to know when that will occur. The **with** statement in this role is an alternative that allows us to be sure that the close will occur after execution of a specific block of code.