**Q1. What is the purpose of the try statement?**

The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

Example

Print one message if the try block raises a NameError and another for other errors:

**try:  
  print(x)  
except NameError:  
  print("Variable x is not defined")  
except:  
  print("Something else went wrong")**

The **try** block lets you test a block of code for errors.

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

The Different Try/Except Variations. So far we've used a try / except and even a try / except / except , but this is only two-thirds of the story. There are two other optional segments to a try block: else and finally . Both of these optional blocks will come after the try and the except .

# **The Different Try/Except Variations**

So far we’ve used a try/except and even a try/except/except, but this is only two-thirds of the story.

There are two other optional segments to a try block: else and finally. Both of these optional blocks will come **after** the try and the except. Also, there’s nothing stopping you from using both else and finally in a single statement — but keep them in that order if you do.

Let’s go through each individually and see how they extend the behavior of a simple try/except.

## Try/Except/Else

When attaching an else statement to the end of a try/except, this code will be executed **after** the try has been completed, but only **if no exceptions occur**.

We can take the previous example of prompting a user for an integer input and use an else block to thank them for valid input and breaking out of the while loop.

while True:  
 try:  
 num = int(input("Enter an int: "))  
 except Exception as e:  
 print(e)  
 else:  
 print("Thank you for the integer!")  
 break# Enter an int: a  
# invalid literal for int() with base 10: 'a'  
# Enter an int: 3  
# Thank you for the integer

## **Try/Except/Finally**

When attaching a finally statement to the end of a try/except, this code will be executed **after** the try has been completed, **regardless of exceptions**.

Again, we’ll use our previous example and add a simple counter to illustrate this behavior.

count = 0  
while True:  
 try:  
 num = int(input("Enter an int: "))  
 break  
 except Exception as e:  
 print(e)  
 finally:  
 count += 1  
 print("Attempt #:",count)# Enter an int: a  
# invalid literal for int() with base 10: 'a'  
# Attempt #: 1  
# Enter an int: 3  
# Attempt #: 2

This might look a bit odd because the break is still inside the try. It’s reasonable to think that the finally would be cut short upon proper input, however, that’s not the case. The finally section will still execute, regardless of how the try is exited.

**Q3. What is the purpose of the raise statement?**

The raise keyword is used **to raise an exception**. You can define what kind of error to raise, and the text to print to the user.

Example

Raise an error and stop the program if x is lower than 0:

x = -1  
  
if x < 0:  
  raise Exception("Sorry, no numbers below zero")

## **Definition and Usage**

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user

Raise a TypeError if x is not an integer:

x = "hello"  
  
if not type(x) is int:  
  raise TypeError("Only integers are allowed")

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

The assert keyword is used when debugging code. The assert keyword **lets you test if a condition in your code returns True, if not, the program will raise an AssertionError**. You can write a message to be written if the code returns False, check the example below.

## **Definition and Usage**

The assert keyword is used when debugging code.

The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.

You can write a message to be written if the code returns False, check the example below.

### Example

Write a message if the condition is False:

x = "hello"  
  
#if condition returns False, AssertionError is raised:  
assert x == "goodbye", "x should be 'hello'"

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

The with statement is a replacement for commonly used try/finally error-handling statements. A common example of using the with statement is opening a file. To open and write to a file in Python, you can use the with statement as follows: with open("example.

The with statement is a replacement for commonly used try/finally error-handling statements. A common example of using the with statement is opening a file. To open and write to a file in Python, you can use the with statement as follows:

**file.write("Hello World!")**

The with statement automatically closes the file after you’ve completed writing it.

Under the hood, the with statement replaces this kind of try-catch block:

f = open("example.txt", "w")

try:

f.write("hello world")

finally:

f.close()

with FileWriter("example.txt") creates a new FileWriter object and calls \_\_enter\_\_().

The \_\_enter\_\_() method initializes the resource you want to use. In this case, it opens a text file. It also has to return the descriptor of the resource, so it returns the opened file.

The as file assigns the file to a variable file.

Finally, the code you want to run with the acquired resource is placed in the with block after the colon.

As soon as this code finishes execution, the \_\_exit\_\_() method is automatically called. In this case, it closes the file.