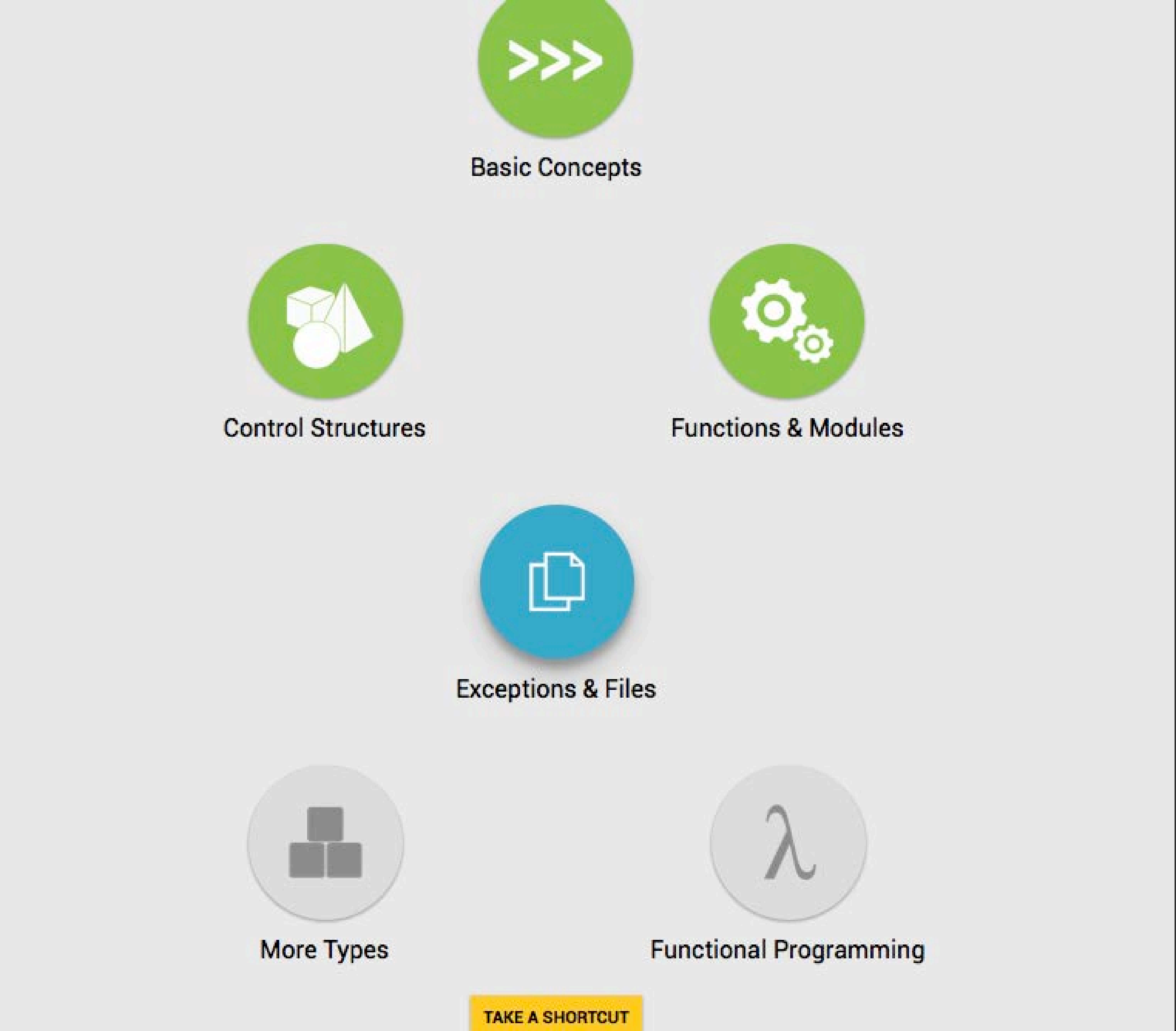
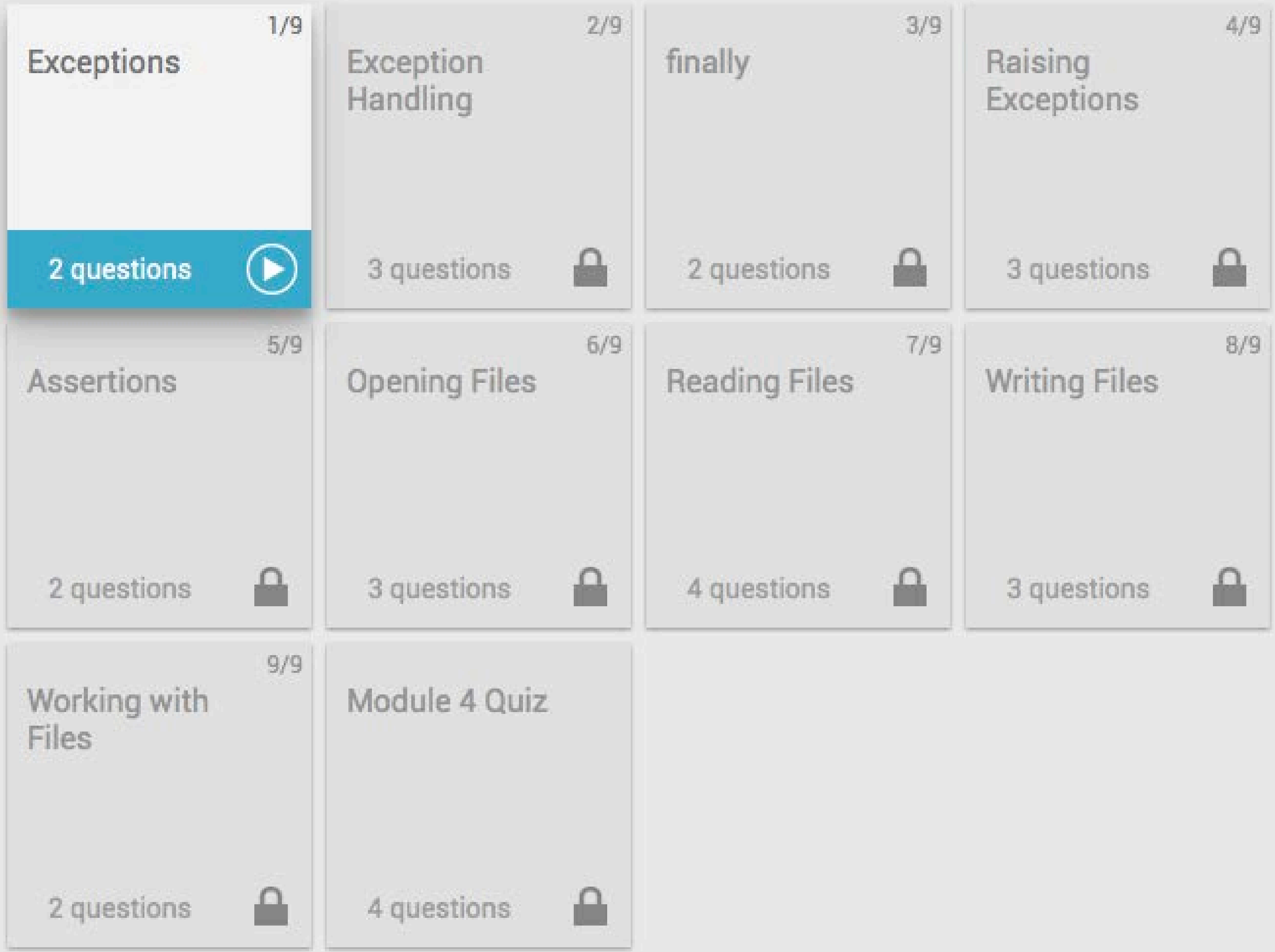

Basic Concepts
Control Structures
Functions & Modules
Exceptions & Files
More Types
Functional Programming
TAKE A SHORTCUT



1/9
Exceptions
2 questions

2/9
Exception Handling
3 questions

3/9
finally
2 questions

4/9
Raising Exceptions
3 questions

5/9
Assertions
2 questions

6/9
Opening Files
3 questions

7/9
Reading Files
4 questions

8/9
Writing Files
3 questions

9/9
Working with Files
2 questions

Module 4 Quiz
4 questions

Exceptions

You have already seen **exceptions** in previous code. They occur when something goes wrong, due to incorrect code or input. When an exception occurs, the program immediately stops.
The following code produces the ZeroDivisionError exception by trying to divide 7 by 0.

```
num1 = 7
num2 = 0
print(num1/num2)
```

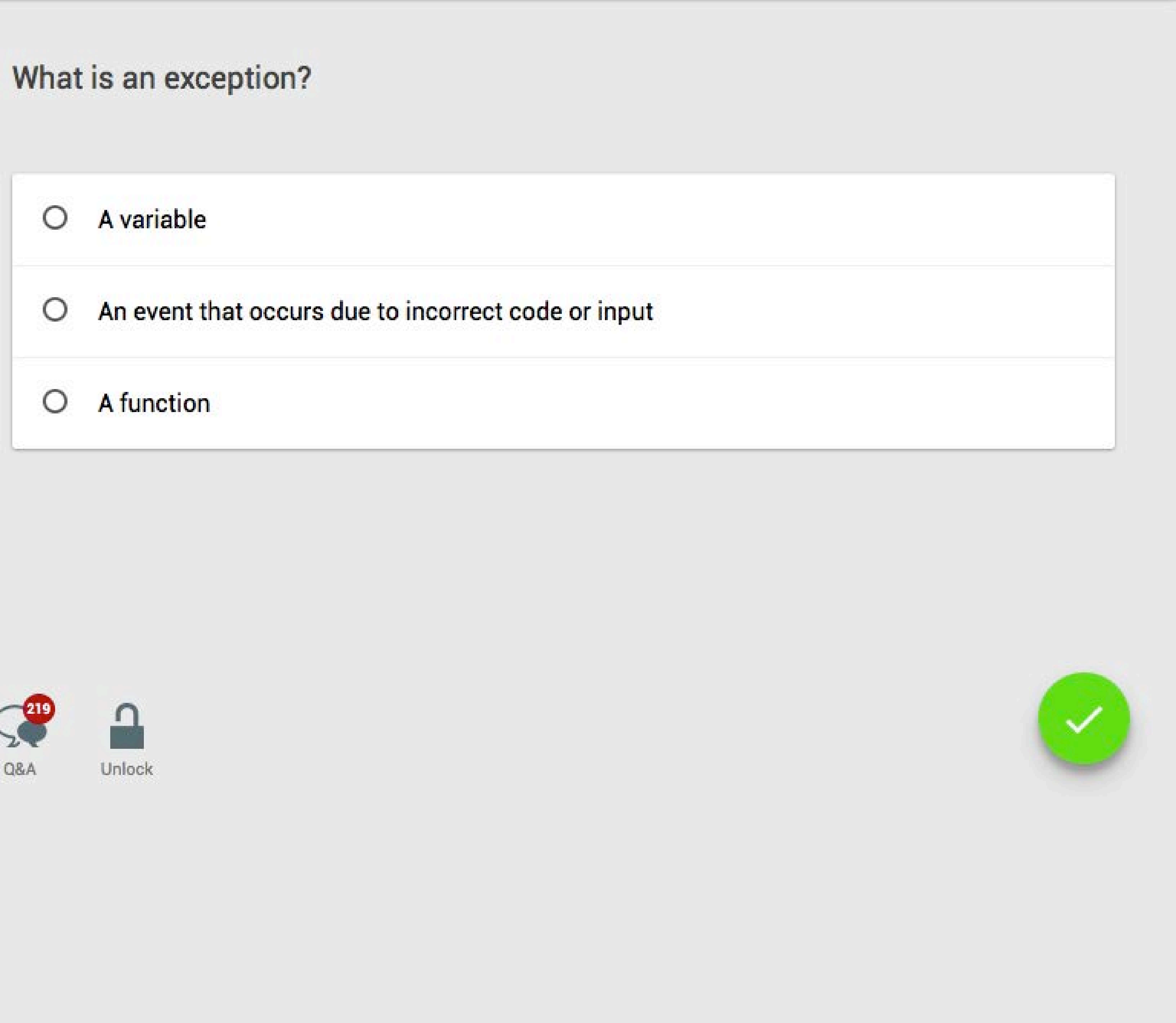
Try It Yourself

Result:

```
>>>
ZeroDivisionError: division by zero
>>>
```

49 COMMENTS

219
Q&A



What is an exception?

- ○ A variable
- ○ An event that occurs due to incorrect code or input
- ○ A function

Exceptions

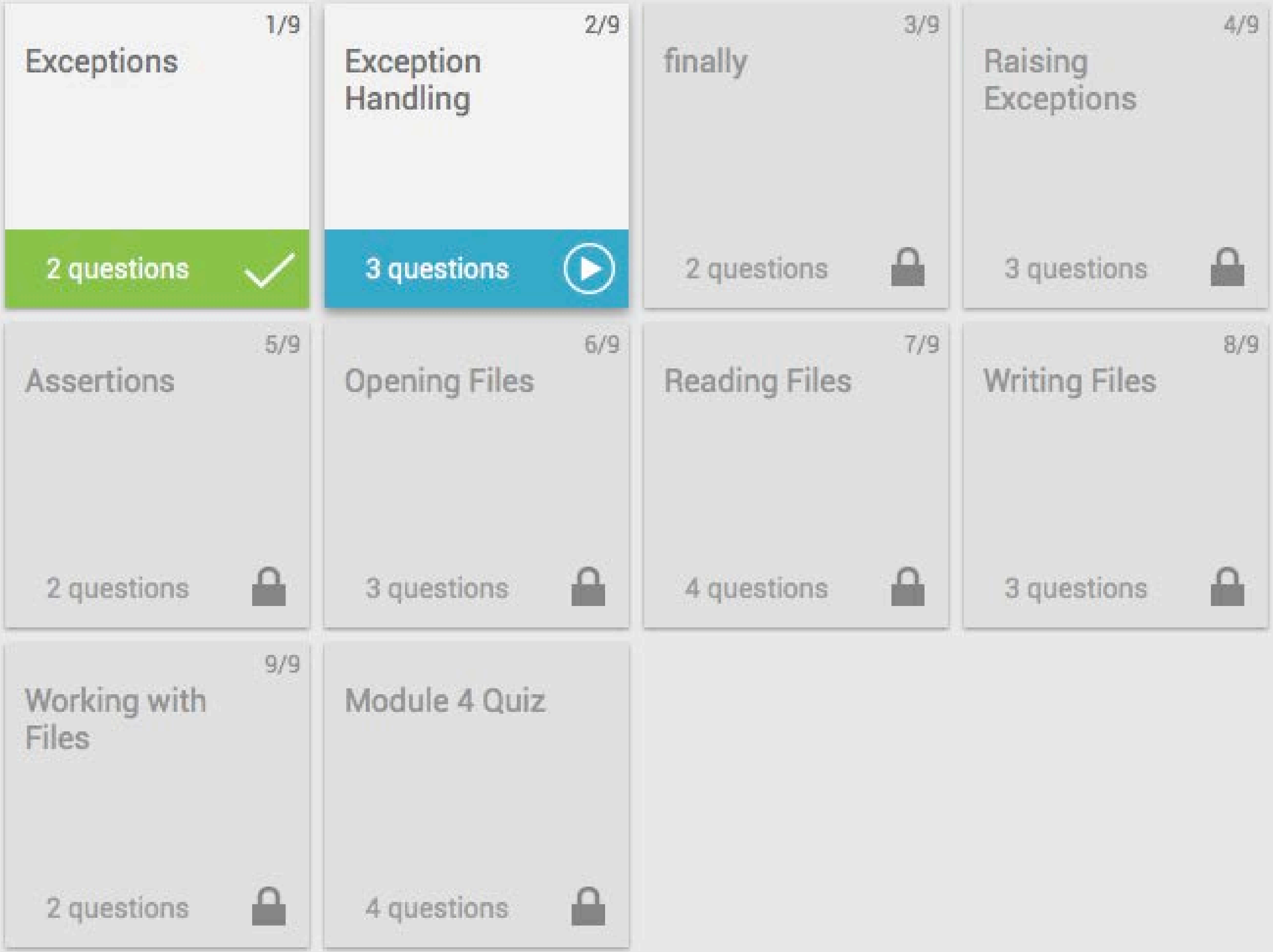
Different exceptions are raised for different reasons.
Common exceptions:
ImportError: an import fails;
IndexError: a list is indexed with an out-of-range number;
NameError: an unknown variable is used;
SyntaxError: the code can't be parsed properly;
TypeError: a function is called on a value of an inappropriate type;
ValueError: a function is called on a value of the correct type, but with an inappropriate value.

> Python has several other built-in exceptions, such as ZeroDivisionError and OSError.
> Third-party libraries also often define their own exceptions.

Which exception is raised by this code?
print("7" + 4)

- ○ ZeroDivisionError
- ○ ValueError
- ○ TypeError

219
Q&A
Unlock



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions

3/9
finally
2 questions

4/9
Raising Exceptions
3 questions

5/9
Assertions
2 questions

6/9
Opening Files
3 questions

7/9
Reading Files
4 questions

8/9
Writing Files
3 questions

9/9
Working with Files
2 questions

Module 4 Quiz
4 questions

Exception Handling

To handle exceptions, and to call code when an exception occurs, you can use a **try/except** statement.
The **try** block contains code that might throw an exception. If that exception occurs, the code in the **try** block stops being executed, and the code in the **except** block is run. If no error occurs, the code in the **except** block doesn't run.
For example:

```
try:
  num1 = 7
  num2 = 0
  print (num1 / num2)
  print("Done calculation")
except ZeroDivisionError:
  print("An error occurred")
  print("due to zero division")
```

Try It Yourself

Result:

```
>>>
An error occurred
due to zero division
>>>
```

In the code above, the except statement defines the type of exception to handle (in our

For example:

```
try:
  num1 = 7
  num2 = 0
  print (num1 / num2)
  print("Done calculation")
except ZeroDivisionError:
  print("An error occurred")
  print("due to zero division")
```

Try It Yourself

Result:

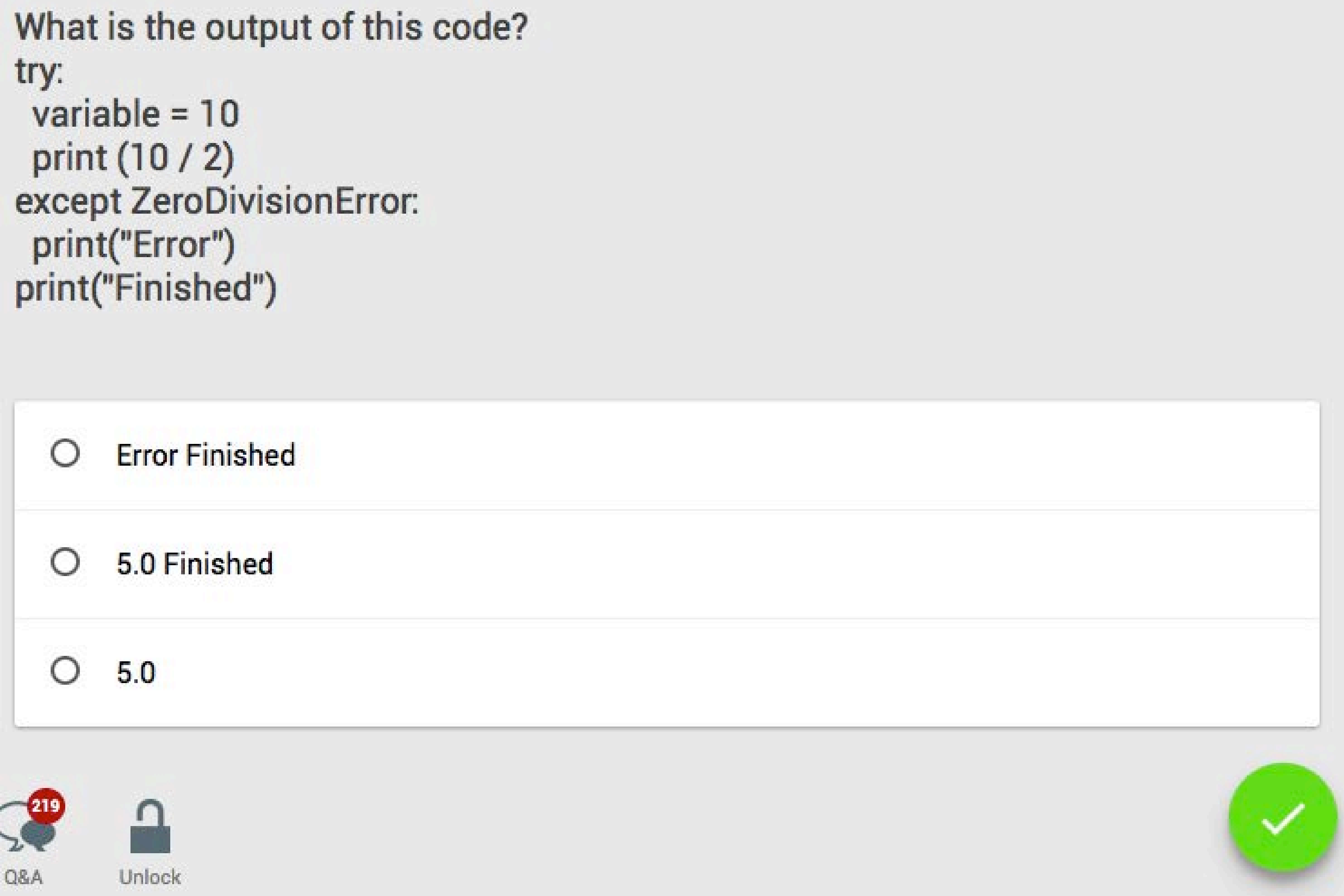
```
>>>
An error occurred
due to zero division
>>>
```

In the code above, the except statement defines the type of exception to handle (in our case, the **ZeroDivisionError**).

80 COMMENTS

219

Q&A



What is the output of this code?

```
try:
  variable = 10
  print (10 / 2)
except ZeroDivisionError:
  print("Error")
print("Finished")
```

- Error Finished
- 5.0 Finished
- 5.0

Exception Handling

A **try** statement can have multiple different **except** blocks to handle different exceptions. Multiple exceptions can also be put into a single **except** block using parentheses, to have the **except** block handle all of them.

```
try:
  variable = 10
  print(variable + "hello")
  print(variable / 2)
except ZeroDivisionError:
  print("Divided by zero")
except (ValueError, TypeError):
  print("Error occurred")
```

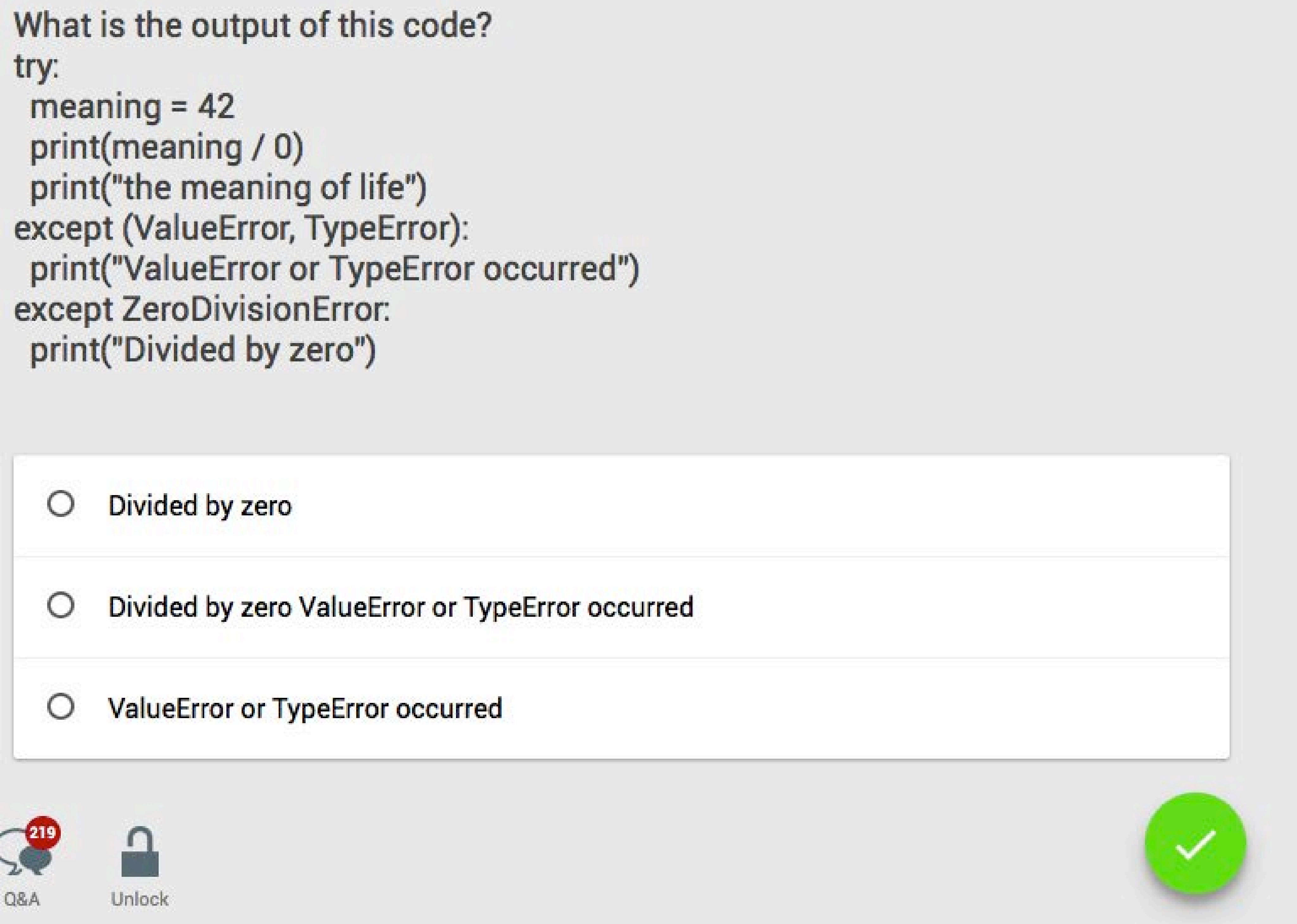
Try It Yourself

Result:

```
>>>
Error occurred
>>>
```

85 COMMENTS

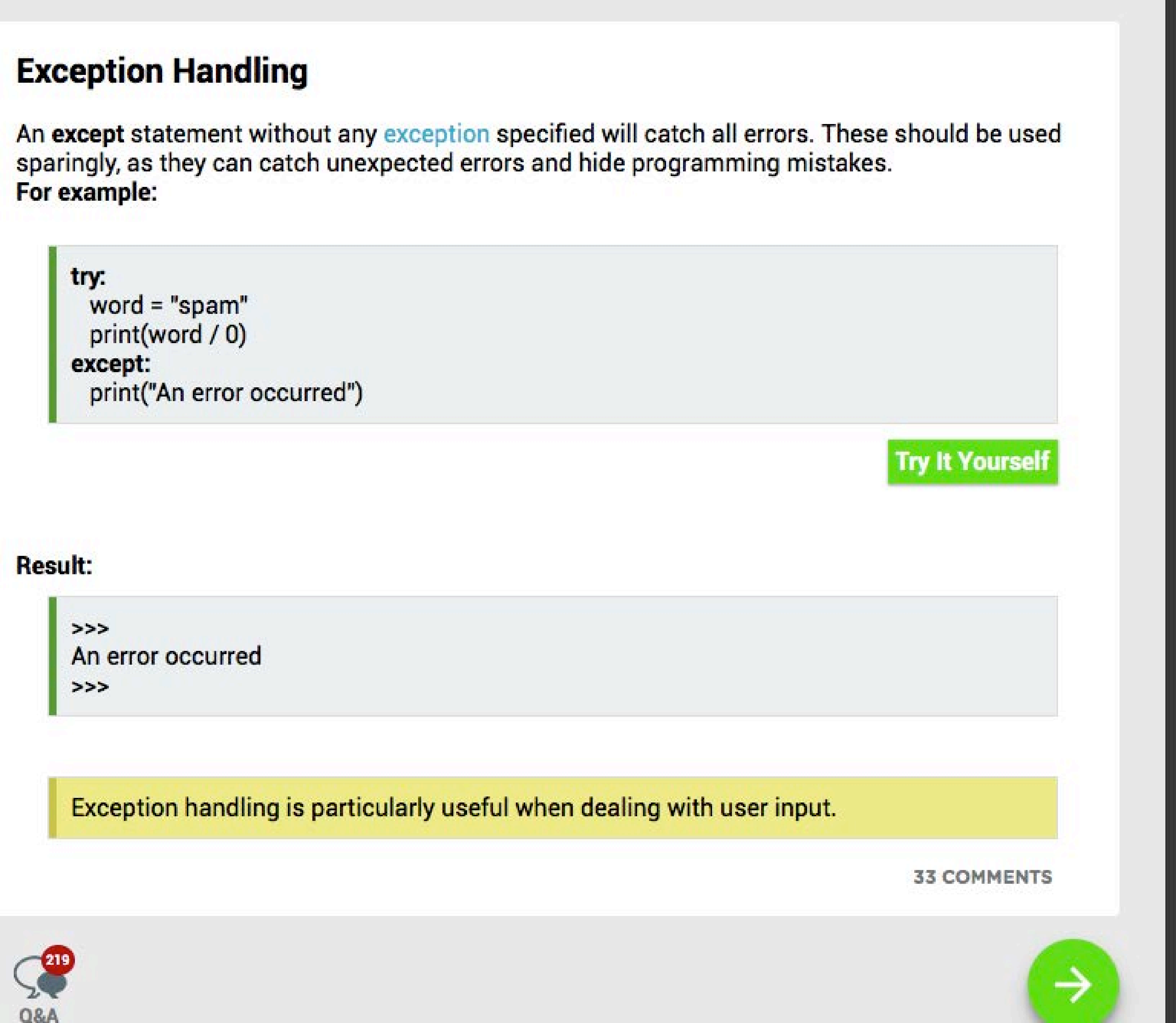
219
Q&A



What is the output of this code?

```
try:
  meaning = 42
  print(meaning / 0)
  print("the meaning of life")
except (ValueError, TypeError):
  print("ValueError or TypeError occurred")
except ZeroDivisionError:
  print("Divided by zero")
```

- ○ Divided by zero
- ○ Divided by zero ValueError or TypeError occurred
- ○ ValueError or TypeError occurred



Exception Handling

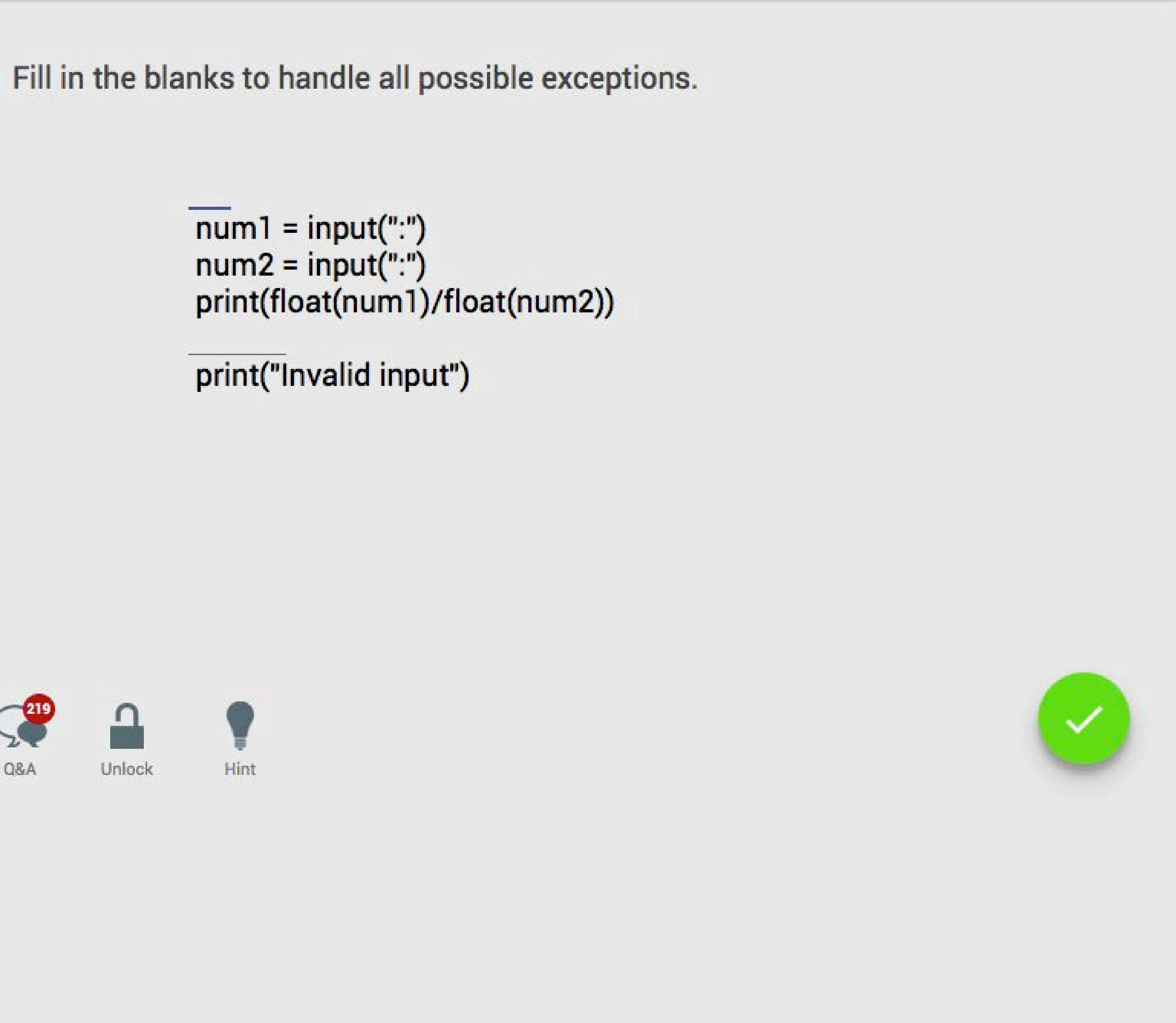
An **except** statement without any exception specified will catch all errors. These should be used sparingly, as they can catch unexpected errors and hide programming mistakes.
For example:

```
try:
  word = "spam"
  print(word / 0)
except:
  print("An error occurred")
```

Result:

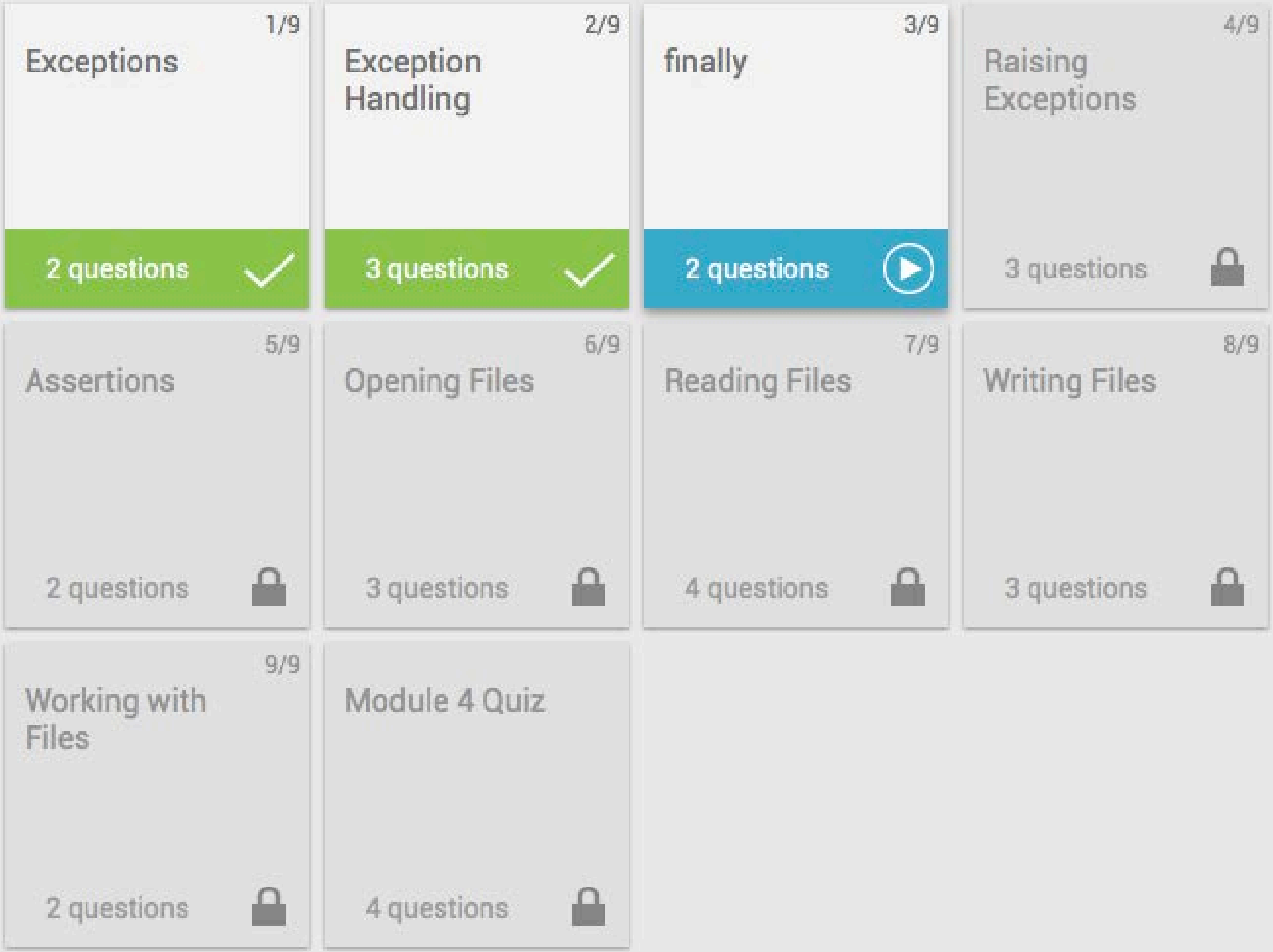
```
>>>
An error occurred
>>>
```

Exception handling is particularly useful when dealing with user input.



Fill in the blanks to handle all possible exceptions.

```
___
num1 = input(":")
num2 = input(":")
print(float(num1)/float(num2))
______
print("Invalid input")
```



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions

4/9
Raising Exceptions
3 questions

5/9
Assertions
2 questions

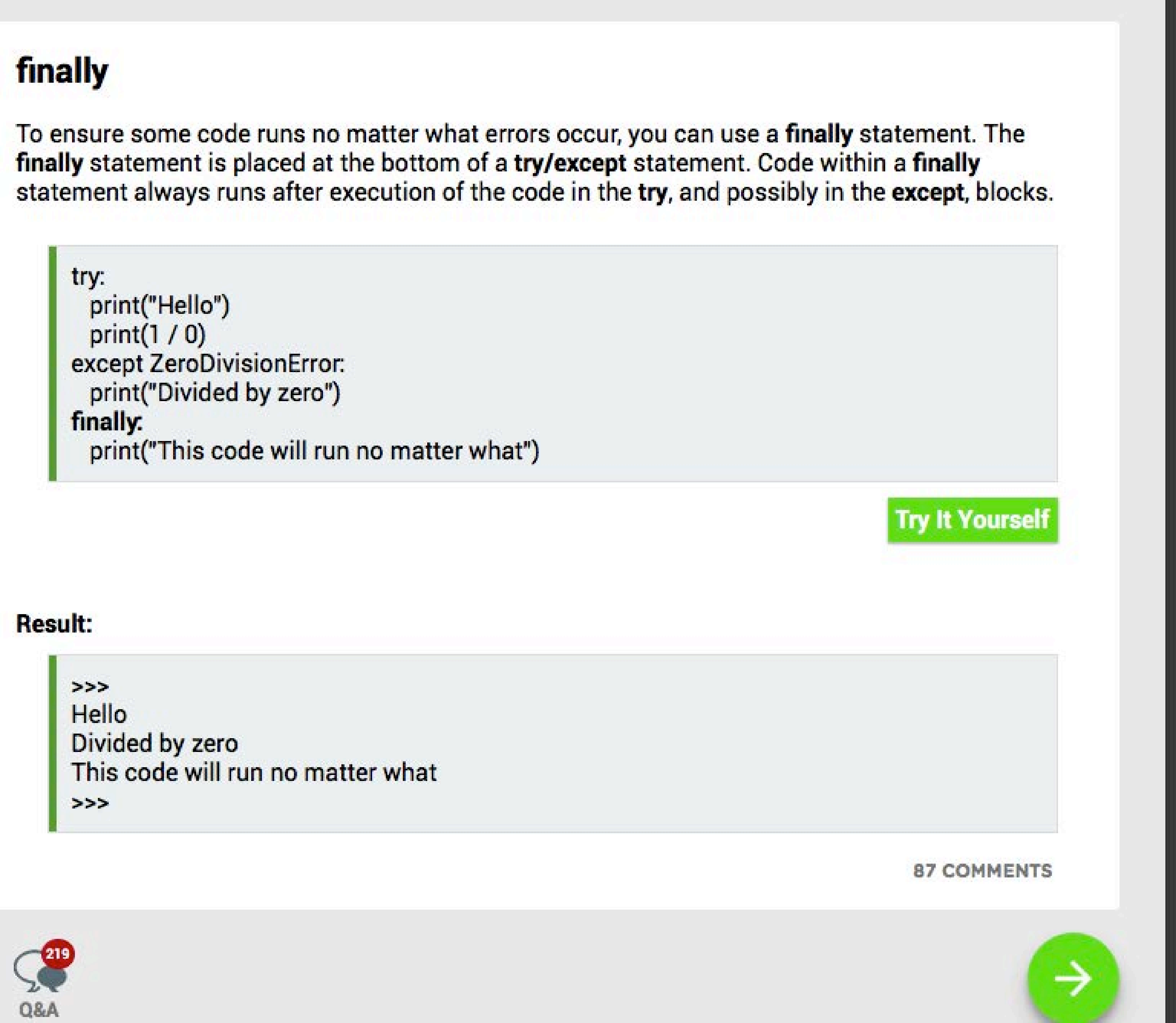
6/9
Opening Files
3 questions

7/9
Reading Files
4 questions

8/9
Writing Files
3 questions

9/9
Working with Files
2 questions

Module 4 Quiz
4 questions



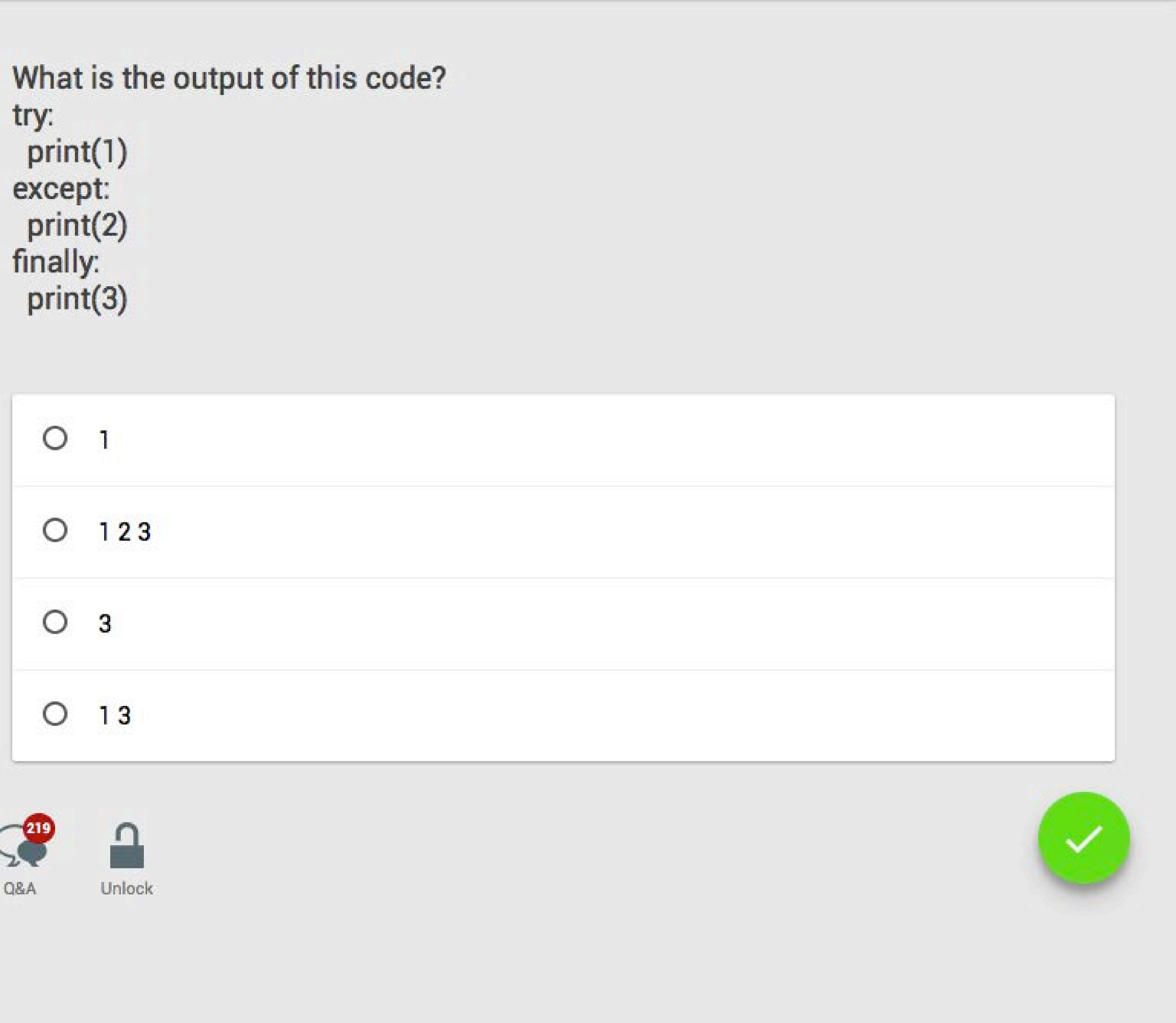
finally

To ensure some code runs no matter what errors occur, you can use a **finally** statement. The **finally** statement is placed at the bottom of a **try/except** statement. Code within a **finally** statement always runs after execution of the code in the **try**, and possibly in the **except**, blocks.

```
try:
   print("Hello")
   print(1 / 0)
except ZeroDivisionError:
   print("Divided by zero")
finally:
   print("This code will run no matter what")
```

Result:

```
>>>
Hello
Divided by zero
This code will run no matter what
>>>
```



What is the output of this code?

```
try:
  print(1)
except:
  print(2)
finally:
  print(3)
```

- ○ 1
- ○ 1 2 3
- ○ 3
- ○ 1 3

finally

Code in a **finally** statement even runs if an uncaught exception occurs in one of the preceding blocks.

```
try:
  print(1)
  print(10 / 0)
except ZeroDivisionError:
  print(unknown_var)
finally:
  print("This is executed last")
```

Try It Yourself

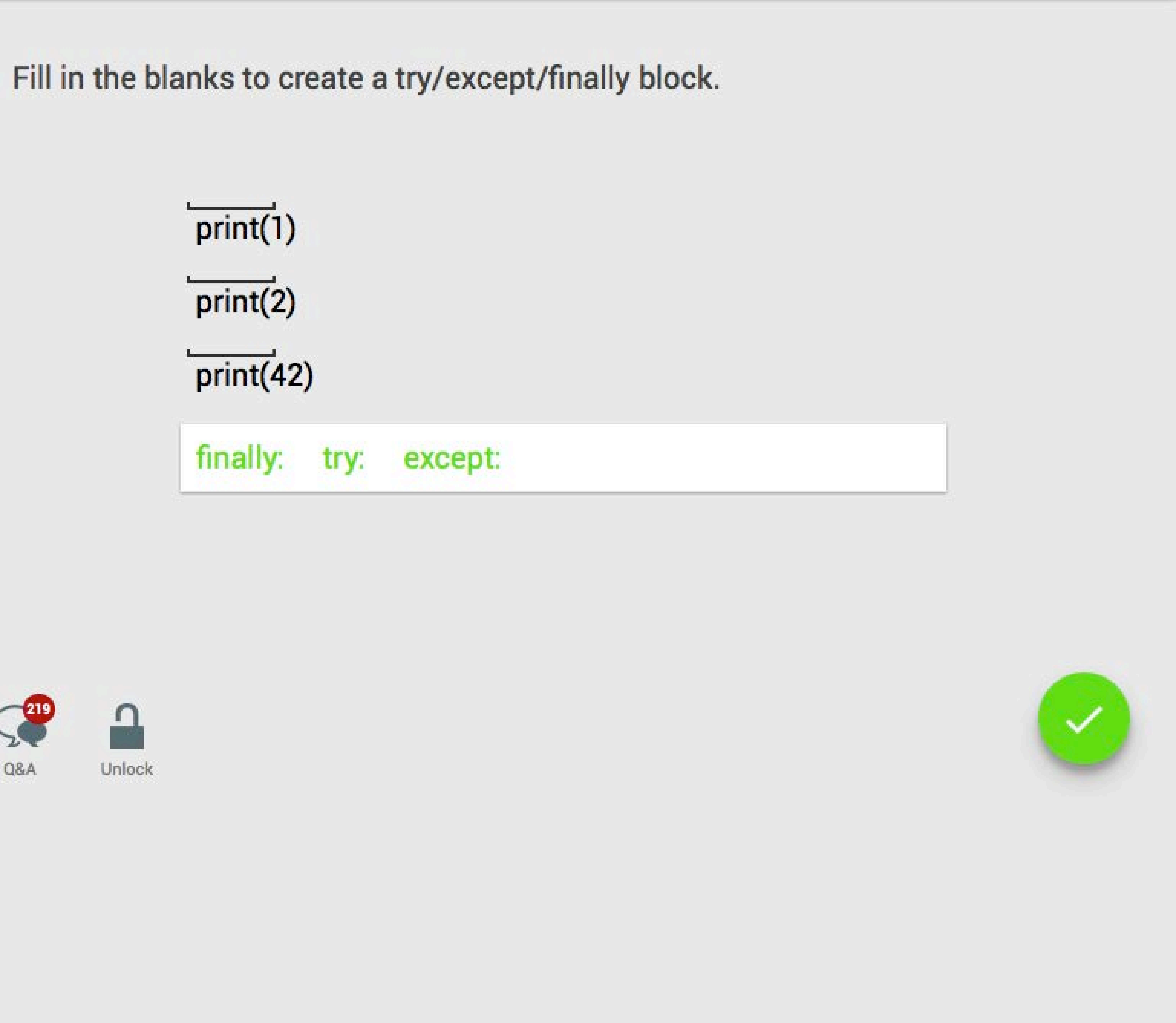
Result:

```
>>>
1
This is executed last

ZeroDivisionError: division by zero
During handling of the above exception, another exception occurred:
NameError: name 'unknown_var' is not defined
>>>
```

82 COMMENTS

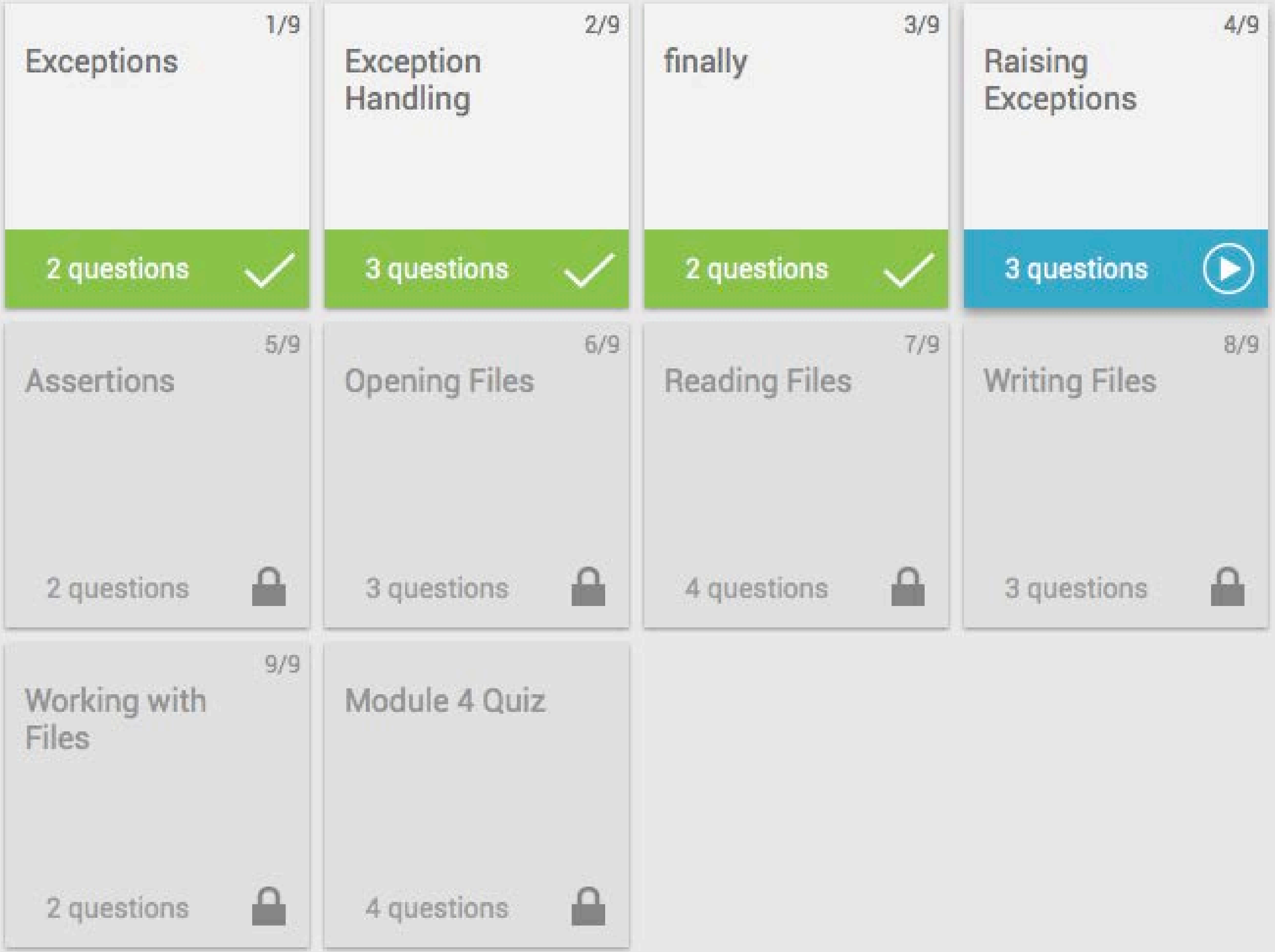
219



Fill in the blanks to create a try/except/finally block.

```
______
print(1)
______
print(2)
______
print(42)
```

finally: try: except:



1/9 Exceptions — 2 questions ✓	2/9 Exception Handling — 3 questions ✓	3/9 finally — 2 questions ✓	4/9 Raising Exceptions — 3 questions ▶
5/9 Assertions — 2 questions 🔒	6/9 Opening Files — 3 questions 🔒	7/9 Reading Files — 4 questions 🔒	8/9 Writing Files — 3 questions 🔒
9/9 Working with Files — 2 questions 🔒	Module 4 Quiz — 4 questions 🔒		

Raising Exceptions

You can raise exceptions by using the **raise** statement.

```
print(1)
raise ValueError
print(2)
```

Try It Yourself

Result:

```
>>>
1
ValueError
>>>
```

You need to specify the **type** of the exception raised.

137 COMMENTS

219
Q&A

Which errors occur in this code?

```
try:
  print(1 / 0)
except ZeroDivisionError:
  raise ValueError
```

- ○ none
- ○ ZeroDivisionError
- ○ ValueError
- ○ ZeroDivisionError and ValueError

219 Q&A   Unlock

Raising Exceptions

Exceptions can be raised with arguments that give detail about them.
For example:

```
name = "123"
raise NameError("Invalid name!")
```

Try It Yourself

Result:

```
>>>
NameError: Invalid name!
>>>
```

65 COMMENTS

219
Q&A

Fill in the blanks to raise a ValueError exception, if the input is negative.

```
num = input(":")
if float(num) _ 0:
  _____ ValueError("Negative!")
```

219 Q&A   Unlock   Hint

Raising Exceptions

In **except** blocks, the **raise** statement can be used without arguments to re-raise whatever exception occurred.
For example:

```
try:
  num = 5 / 0
except:
  print("An error occurred")
  raise
```

Try It Yourself

Result:

```
>>>
An error occurred

ZeroDivisionError: division by zero
>>>
```

70 COMMENTS

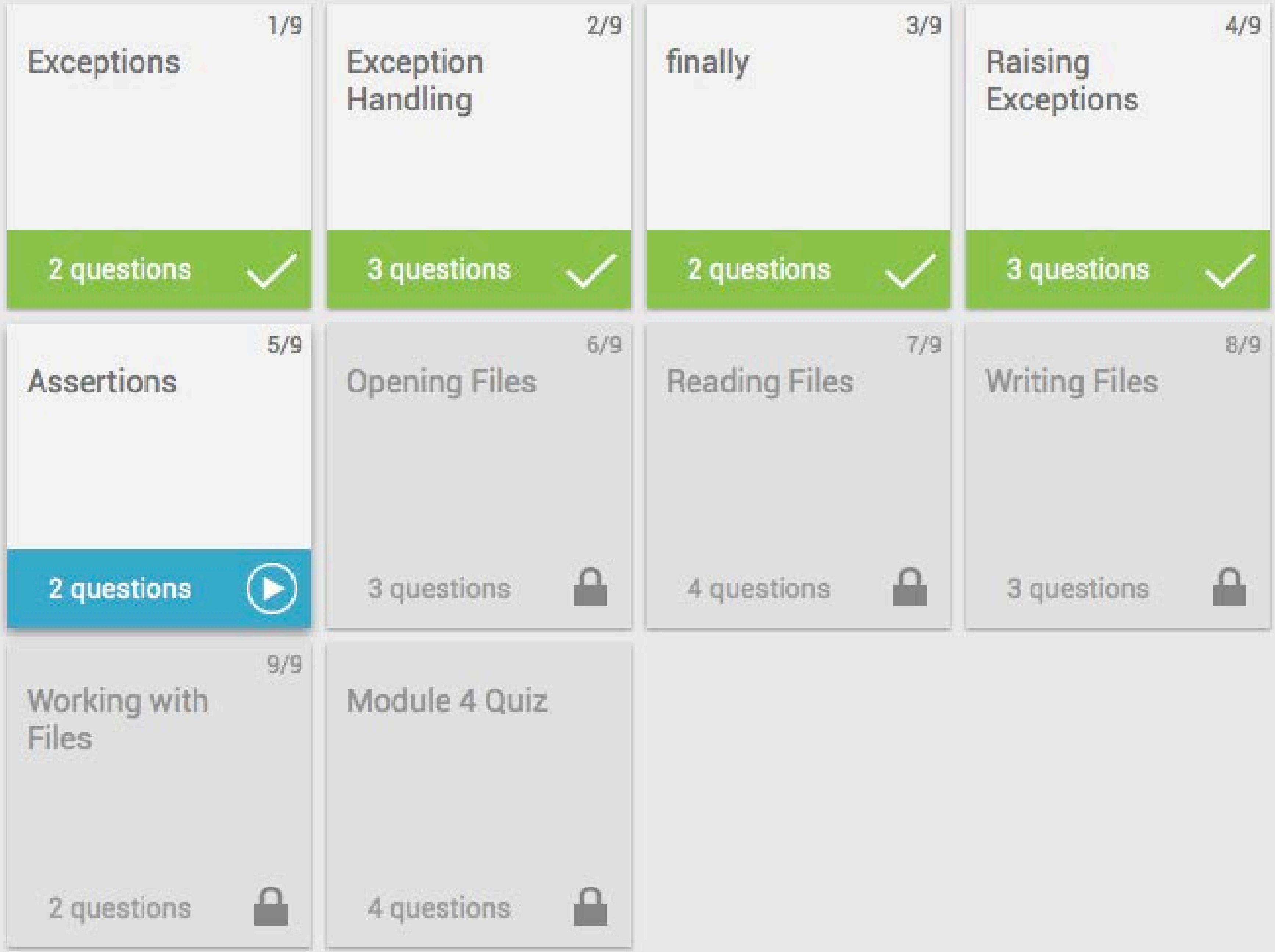
219
Q&A

Can you use the raise statement outside the except block?

- ○ Yes
- ○ No

219 Q&A

Unlock



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions ✓

4/9
Raising Exceptions
3 questions ✓

5/9
Assertions
2 questions ▶

6/9
Opening Files
3 questions 🔒

7/9
Reading Files
4 questions 🔒

8/9
Writing Files
3 questions 🔒

9/9
Working with Files
2 questions 🔒

Module 4 Quiz
4 questions 🔒

Assertions

An **assertion** is a sanity-check that you can turn on or turn off when you have finished testing the program.
An expression is tested, and if the result comes up false, an exception is raised.
Assertions are carried out through use of the **assert** statement.

```
print(1)
assert 2 + 2 == 4
print(2)
assert 1 + 1 == 3
print(3)
```

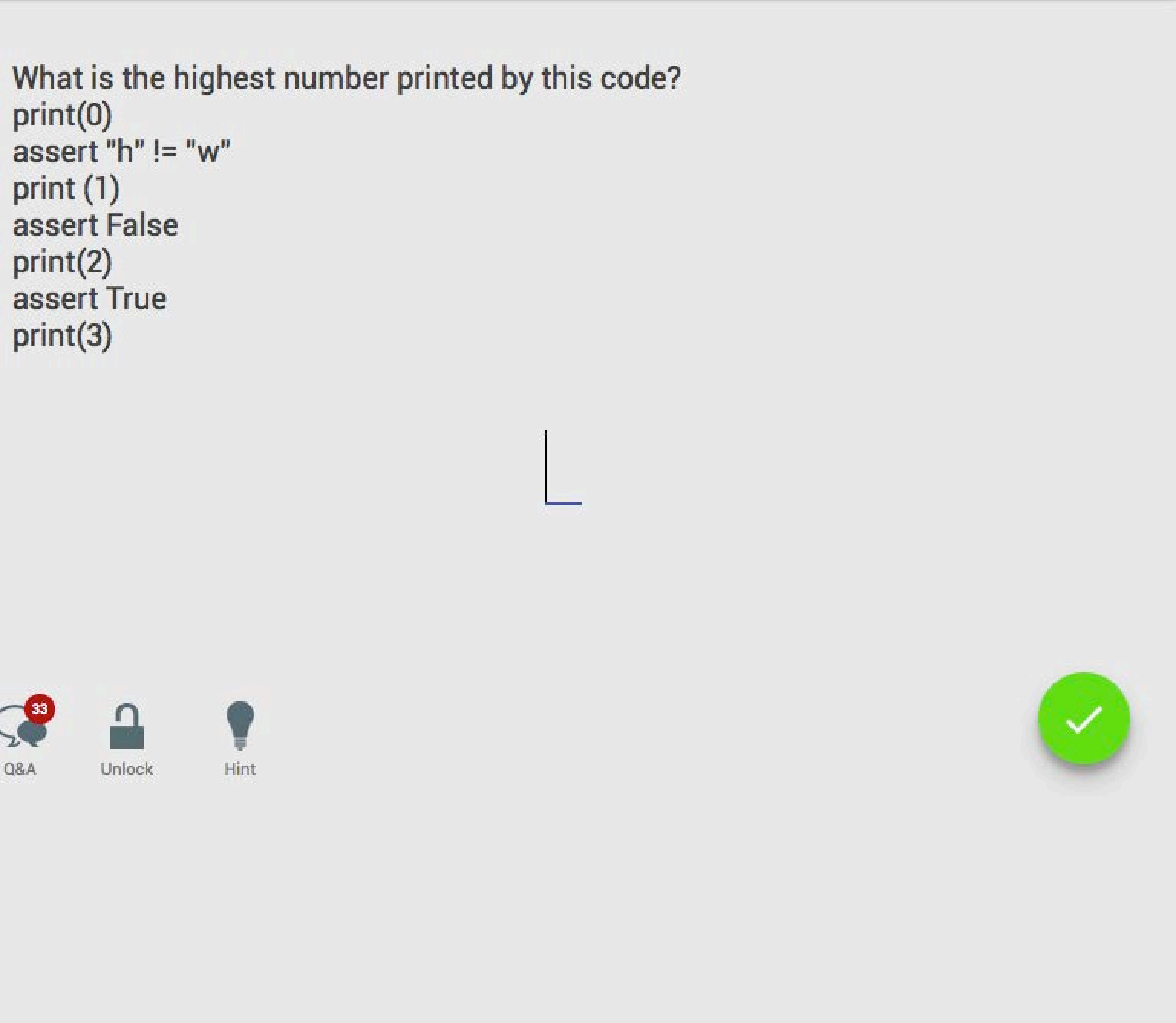
Try It Yourself

Result:

```
>>>
1
2
AssertionError
>>>
```

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

83 COMMENTS



What is the highest number printed by this code?

```
print(0)
assert "h" != "w"
print (1)
assert False
print(2)
assert True
print(3)
```

Assertions

The **assert** can take a second argument that is passed to the AssertionError raised if the assertion fails.

```
temp = -10
assert (temp >= 0), "Colder than absolute zero!"
```

Try It Yourself

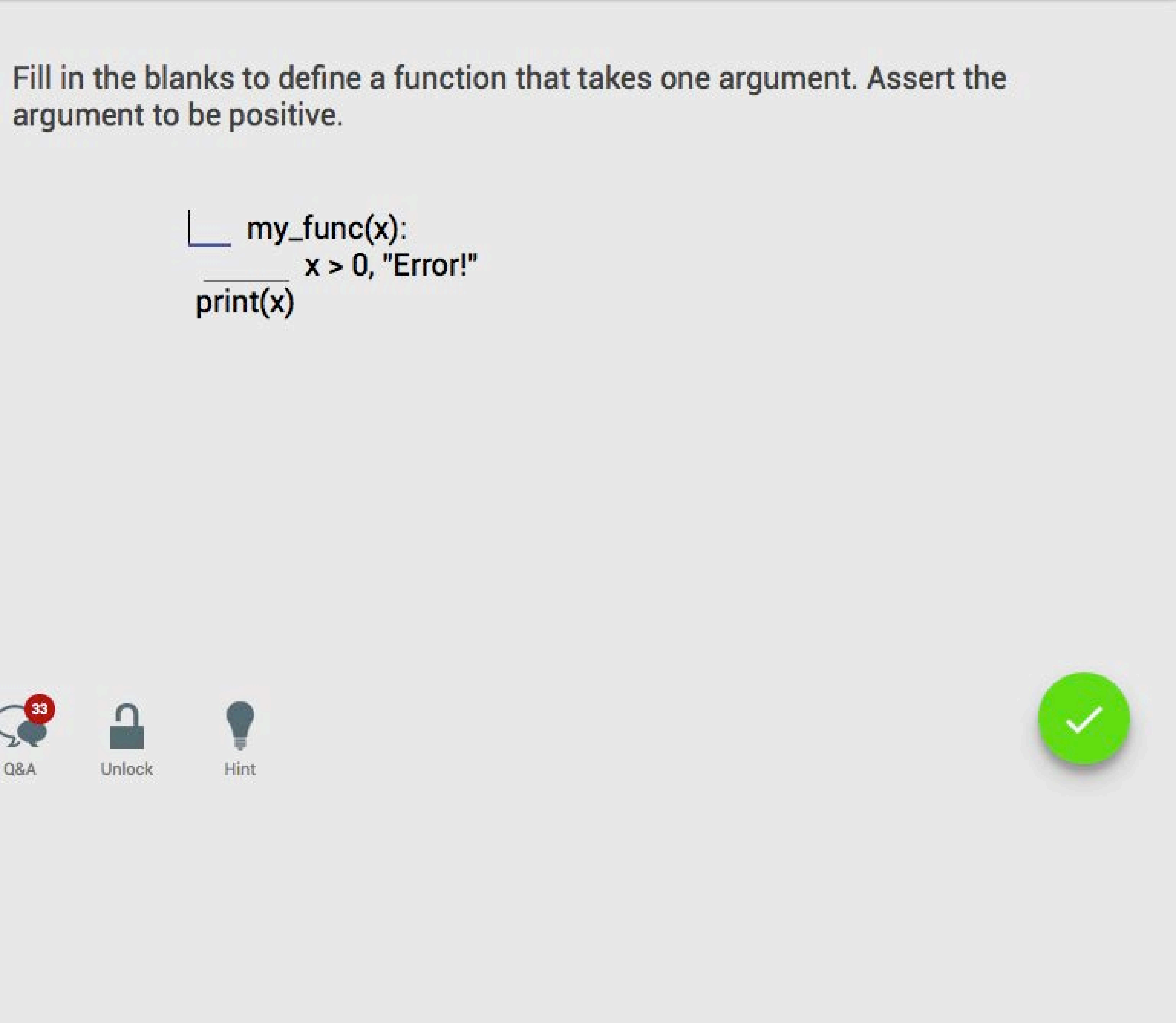
Result:

```
>>>
AssertionError: Colder than absolute zero!
>>>
```

AssertionError exceptions can be caught and handled like any other exception using the **try-except** statement, but if not handled, this type of exception will terminate the program.

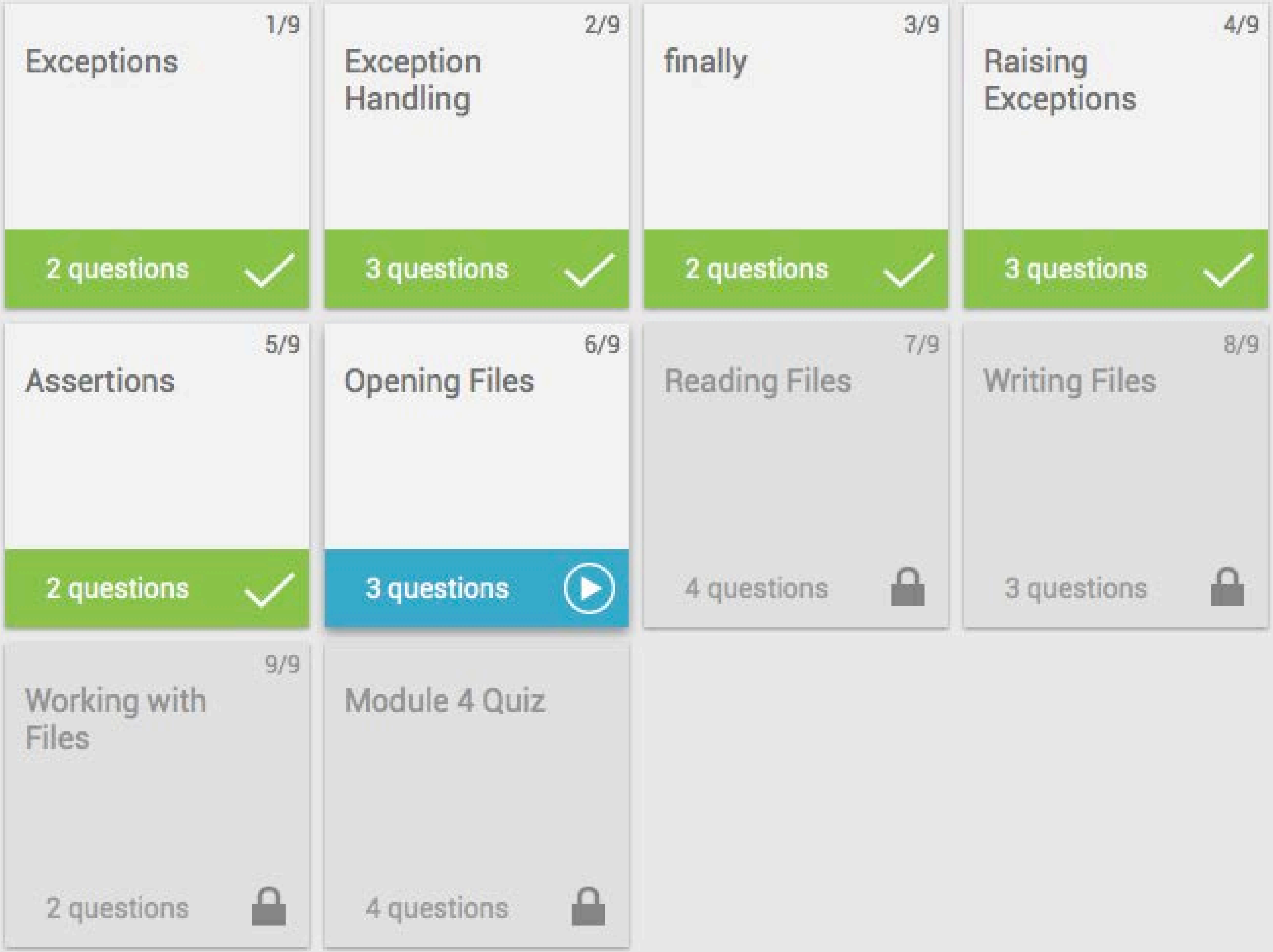
82 COMMENTS

33
Q&A



Fill in the blanks to define a function that takes one argument. Assert the argument to be positive.

```
___ my_func(x):
  ______ x > 0, "Error!"
print(x)
```



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions ✓

4/9
Raising Exceptions
3 questions ✓

5/9
Assertions
2 questions ✓

6/9
Opening Files
3 questions

7/9
Reading Files
4 questions

8/9
Writing Files
3 questions

9/9
Working with Files
2 questions

Module 4 Quiz
4 questions

Opening Files

You can use Python to read and write the contents of **files**.
Text files are the easiest to manipulate. Before a file can be edited, it must be opened, using the **open** function.

```
myfile = open("filename.txt")
```

The argument of the **open** function is the **path** to the file. If the file is in the same directory as the program, you can specify only its name.

91 COMMENTS

306
Q&A

Which function is used to access files?

306

Q&A

Unlock

Hint

Opening Files

You can specify the **mode** used to open a file by applying a second argument to the **open** function.
Sending "r" means open in read mode, which is the default.
Sending "w" means write mode, for rewriting the contents of a file.
Sending "a" means append mode, for adding new content to the end of the file.

Adding "b" to a mode opens it in **binary** mode, which is used for non-text files (such as image and sound files).
For example:

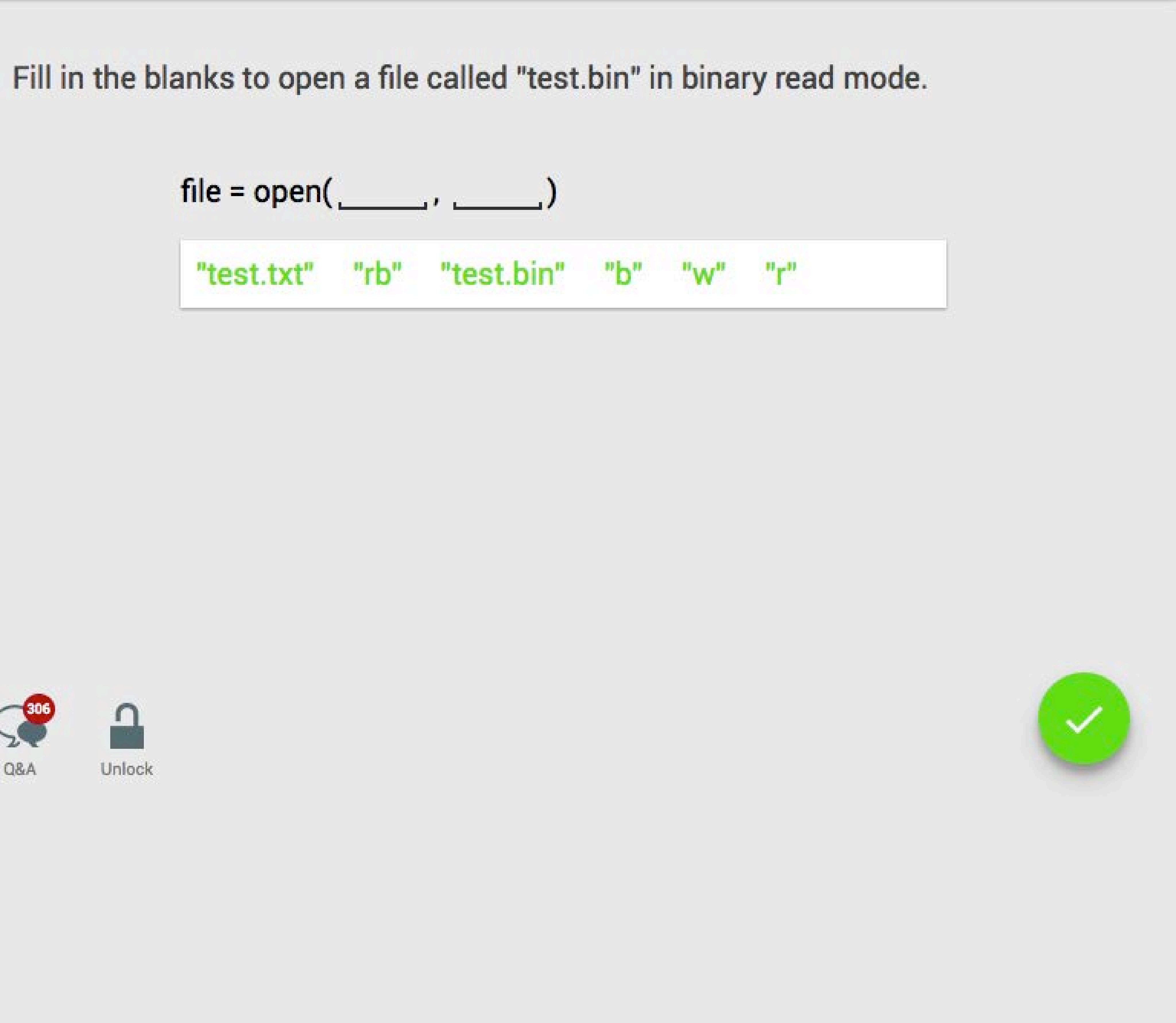
```
# write mode
open("filename.txt", "w")

# read mode
open("filename.txt", "r")
open("filename.txt")

# binary write mode
open("filename.txt", "wb")
```

77 COMMENTS

306
Q&A



Fill in the blanks to open a file called "test.bin" in binary read mode.

```
file = open(_____, _____)
```

"test.txt"   "rb"   "test.bin"   "b"   "w"   "r"

Opening Files

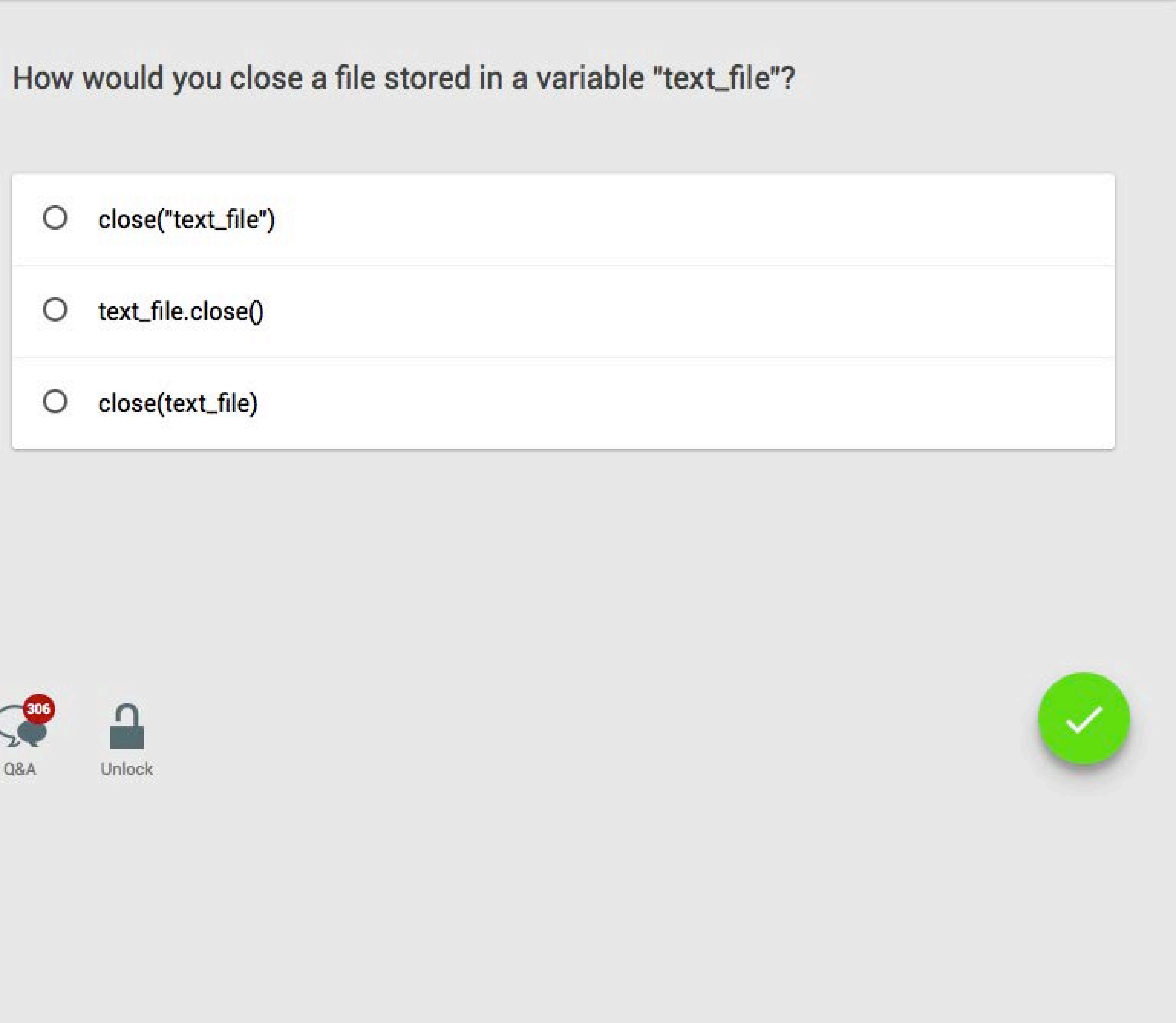
Once a file has been opened and used, you should close it.
This is done with the **close** method of the file object.

```
file = open("filename.txt", "w")
# do stuff to the file
file.close()
```

We will read/write content to files in the upcoming lessons.

51 COMMENTS

306
Q&A

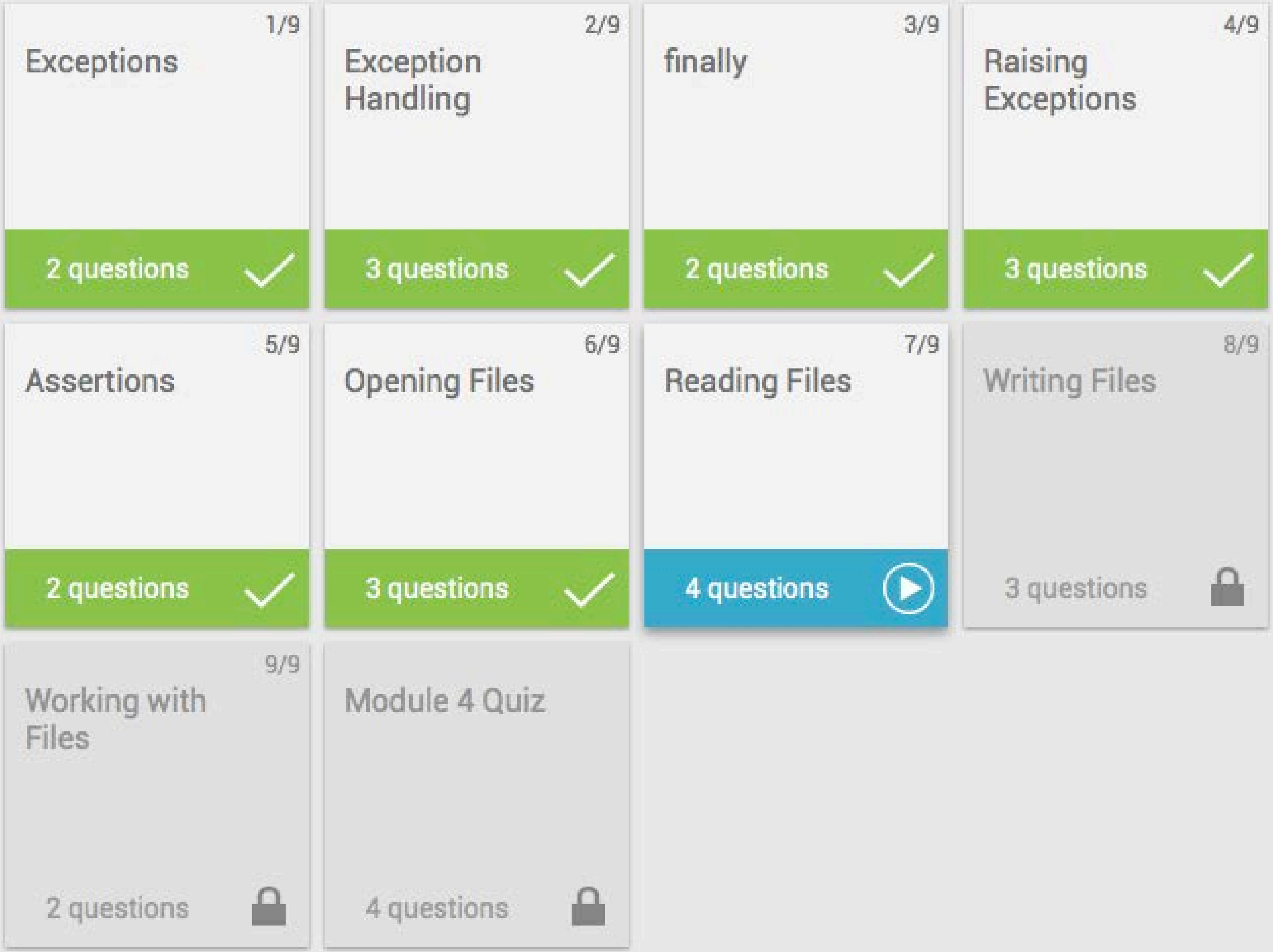


How would you close a file stored in a variable "text_file"?

○ close("text_file")

○ text_file.close()

○ close(text_file)



1/9 Exceptions — 2 questions ✓

2/9 Exception Handling — 3 questions ✓

3/9 finally — 2 questions ✓

4/9 Raising Exceptions — 3 questions ✓

5/9 Assertions — 2 questions ✓

6/9 Opening Files — 3 questions ✓

7/9 Reading Files — 4 questions

8/9 Writing Files — 3 questions

9/9 Working with Files — 2 questions

Module 4 Quiz — 4 questions

Reading Files

The contents of a file that has been opened in text mode can be read using the **read** method.

```
file = open("filename.txt", "r")
cont = file.read()
print(cont)
file.close()
```

This will print all of the contents of the file "filename.txt".

90 COMMENTS

306

Q&A

Rearrange the code to open a file, read its contents, print them, and then close the file.

```
cont = file.read()
```

```
file = open("test.txt")
```

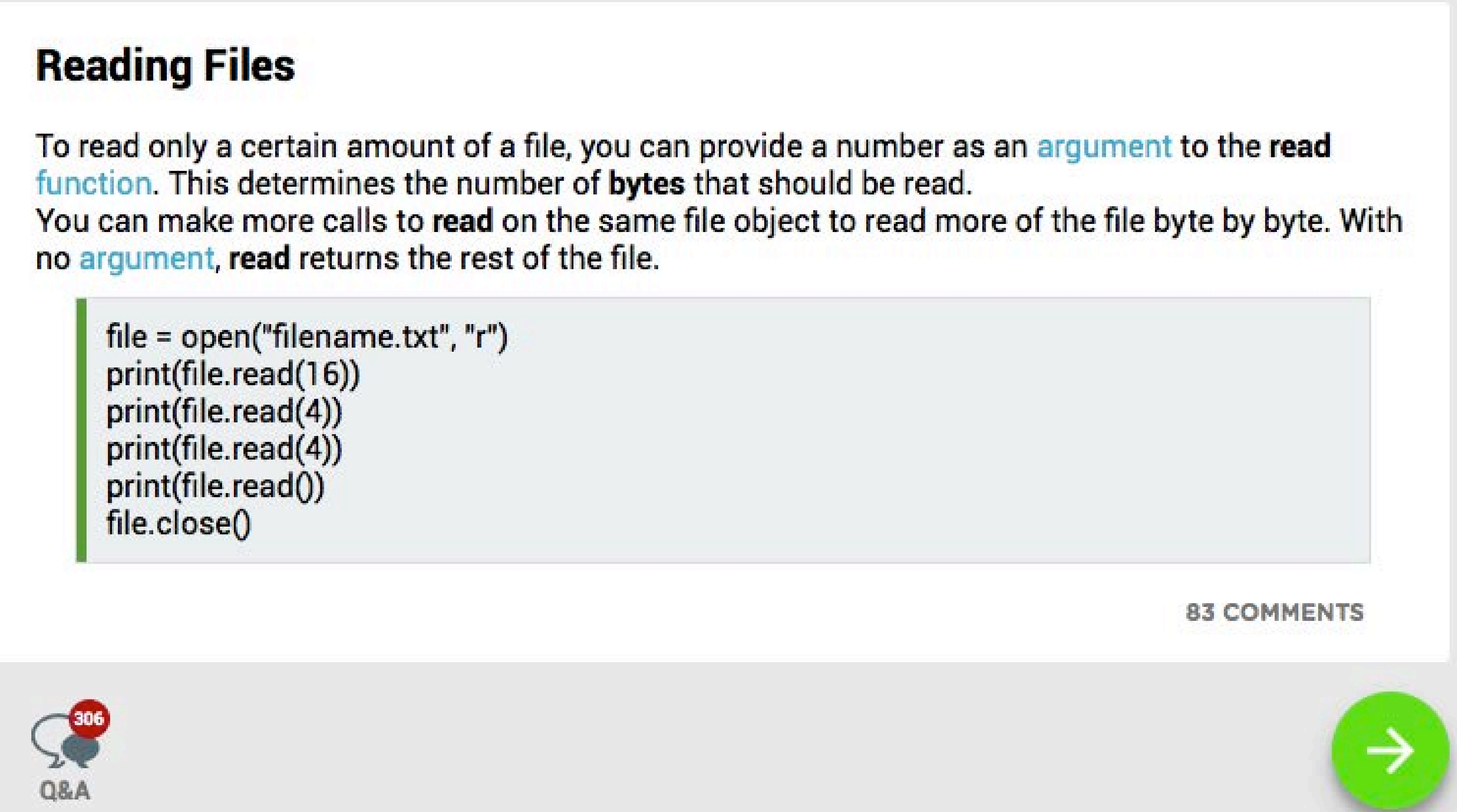
```
file.close()
```

```
print(cont)
```

306

Q&A

Unlock



Reading Files

To read only a certain amount of a file, you can provide a number as an argument to the **read** function. This determines the number of **bytes** that should be read.
You can make more calls to **read** on the same file object to read more of the file byte by byte. With no argument, **read** returns the rest of the file.

```
file = open("filename.txt", "r")
print(file.read(16))
print(file.read(4))
print(file.read(4))
print(file.read())
file.close()
```

How many characters would be in each line printed by this code, if one character is one byte?

```
file = open("filename.txt", "r")
for i in range(21):
  print(file.read(4))
file.close()
```

306

Q&A Unlock Hint

Reading Files

After all contents in a file have been read, any attempts to read further from that file will return an empty string, because you are trying to read from the end of the file.

```
file = open("filename.txt", "r")
file.read()
print("Re-reading")
print(file.read())
print("Finished")
file.close()
```

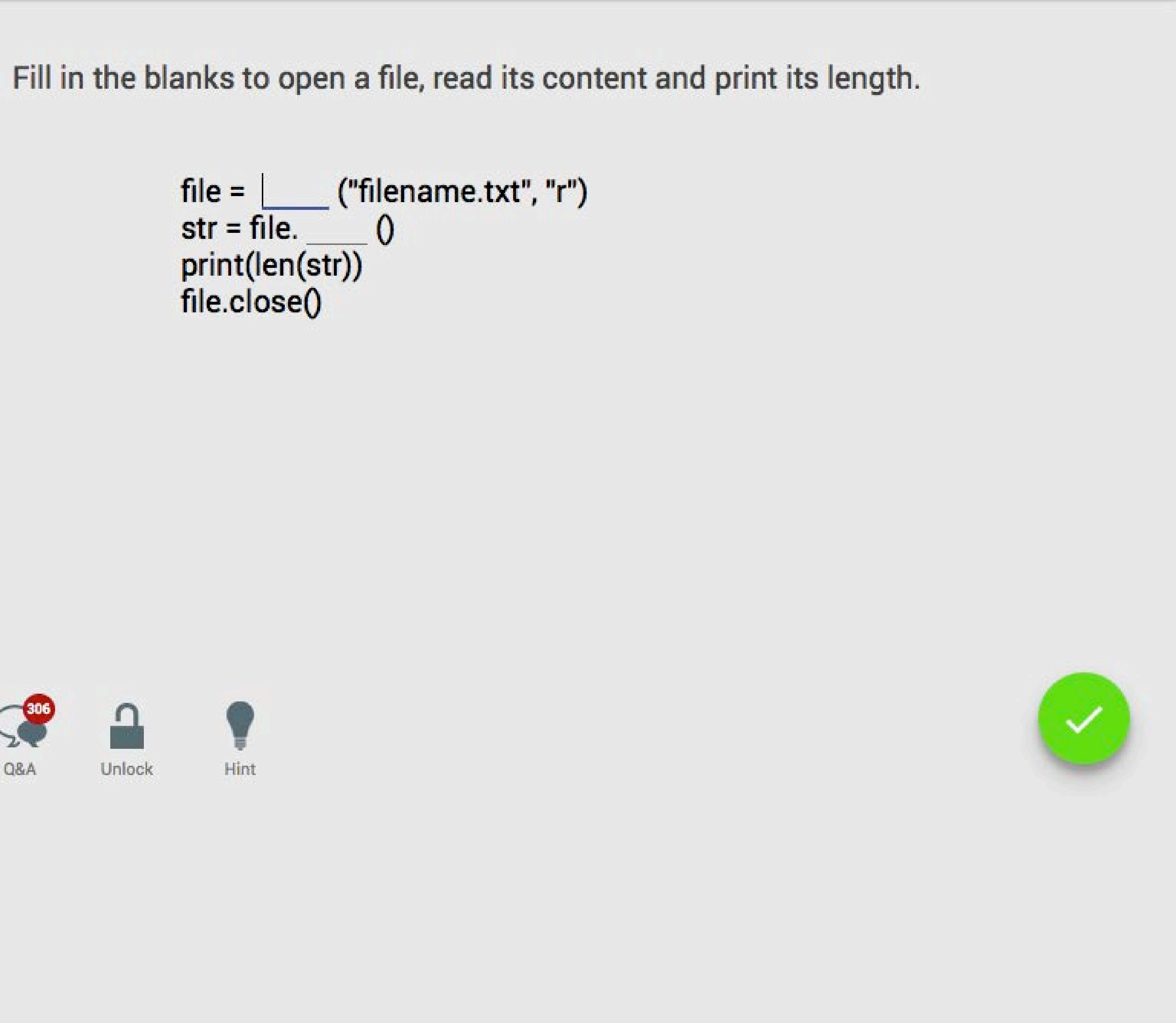
Result:

```
>>>
Re-reading

Finished
>>>
```

52 COMMENTS

306
Q&A



Fill in the blanks to open a file, read its content and print its length.

```
file = ____ ("filename.txt", "r")
str = file. ____ ()
print(len(str))
file.close()
```

Reading Files

To retrieve each line in a file, you can use the **readlines** method to return a list in which each element is a line in the file.
For example:

```
file = open("filename.txt", "r")
print(file.readlines())
file.close()
```

Result:

```
>>>
['Line 1 text \n', 'Line 2 text \n', 'Line 3 text']
>>>
```

You can also use a **for** loop to iterate through the lines in the file:

```
file = open("filename.txt", "r")

for line in file:
   print(line)

file.close()
```

Result:

```
>>>
Line 1 text
```

You can also use a **for** loop to iterate through the lines in the file:

```
file = open("filename.txt", "r")

for line in file:
    print(line)

file.close()
```

Result:

```
>>>
Line 1 text

Line 2 text

Line 3 text
>>>
```

In the output, the lines are separated by blank lines, as the **print** function automatically adds a new line at the end of its output.

102 COMMENTS

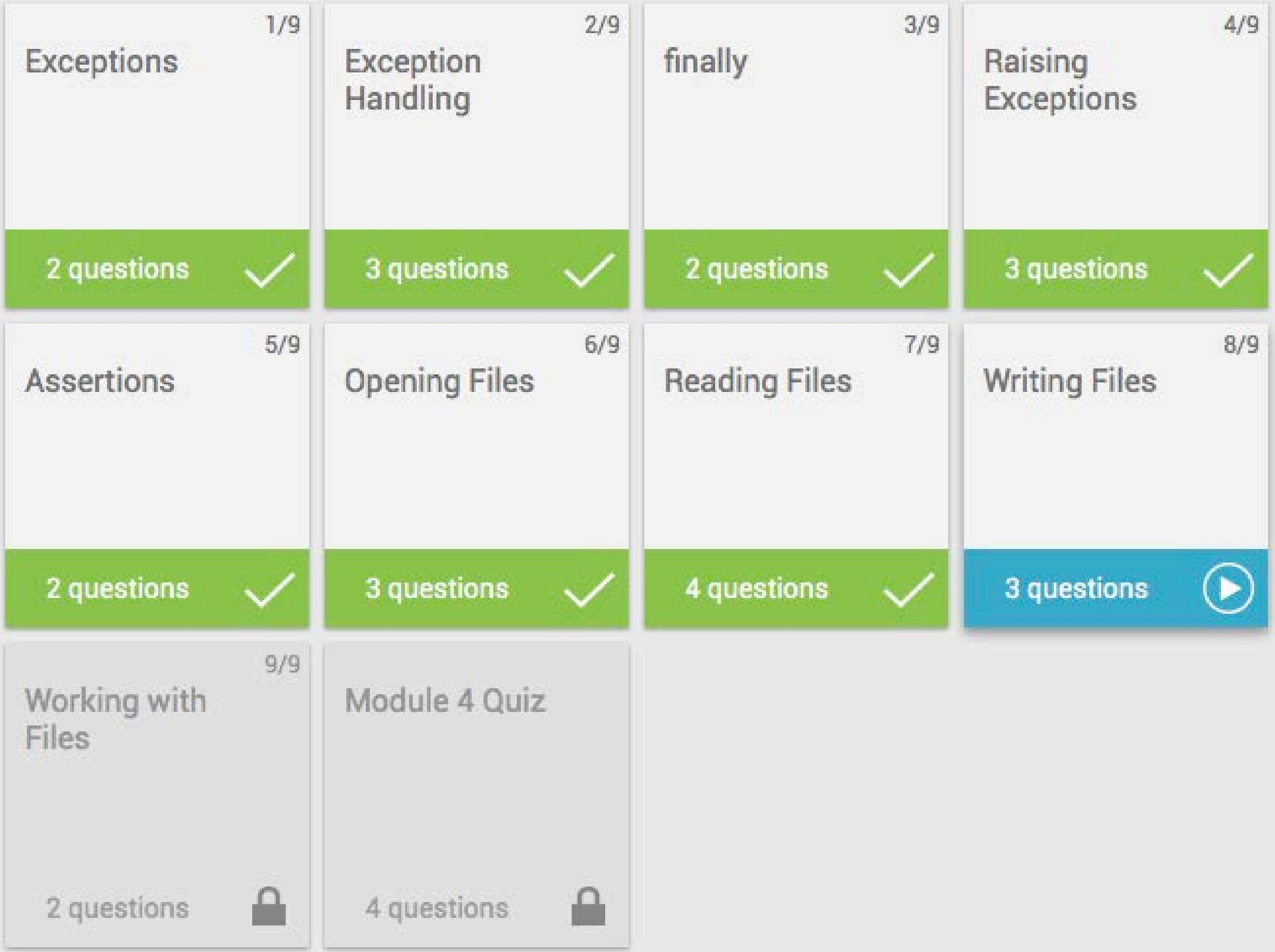
306
Q&A

If the file test.txt has 7 lines of content, what will the following expression return?

```
len(open("test.txt").readlines())
```

306

Q&A Unlock Hint



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions ✓

4/9
Raising Exceptions
3 questions ✓

5/9
Assertions
2 questions ✓

6/9
Opening Files
3 questions ✓

7/9
Reading Files
4 questions ✓

8/9
Writing Files
3 questions ▶

9/9
Working with Files
2 questions 🔒

Module 4 Quiz
4 questions 🔒

Writing Files

To write to files you use the **write** method, which writes a string to the file.
For example:

```
file = open("newfile.txt", "w")
file.write("This has been written to a file")
file.close()

file = open("newfile.txt", "r")
print(file.read())
file.close()
```

Try It Yourself

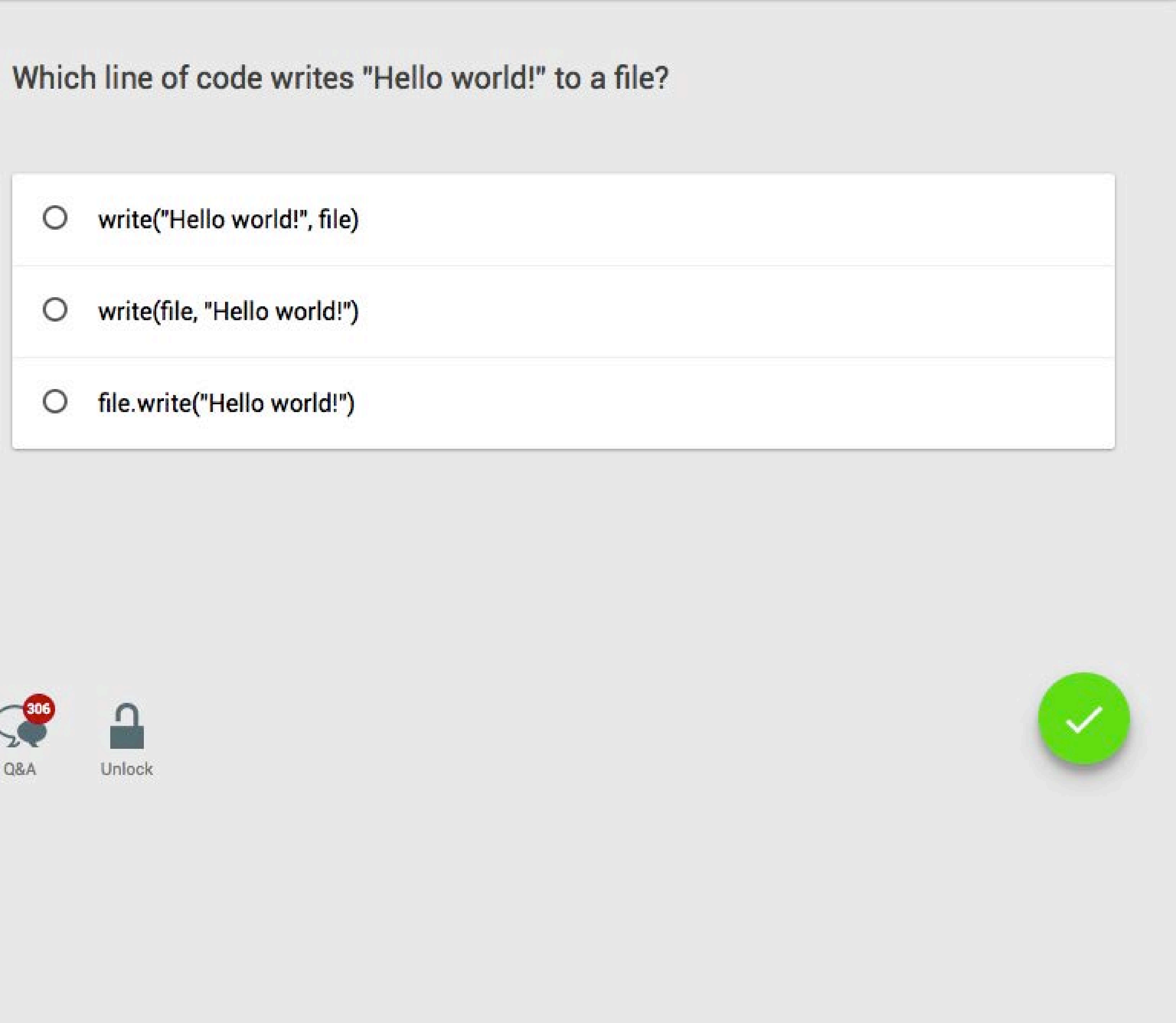
Result:

```
>>>
This has been written to a file
>>>
```

The "w" mode will create a file, if it does not already exist.

93 COMMENTS

306



Which line of code writes "Hello world!" to a file?

- ○ write("Hello world!", file)
- ○ write(file, "Hello world!")
- ○ file.write("Hello world!")

Writing Files

When a file is opened in write mode, the file's existing content is deleted.

```
file = open("newfile.txt", "r")
print("Reading initial contents")
print(file.read())
print("Finished")
file.close()

file = open("newfile.txt", "w")
file.write("Some new text")
file.close()

file = open("newfile.txt", "r")
print("Reading new contents")
print(file.read())
print("Finished")
file.close()
```

Try It Yourself

Result:

```
>>>
Reading initial contents
some initial text
Finished
Reading new contents
Some new text
```

```
file.write("Some new text")
file.close()

file = open("newfile.txt", "r")
print("Reading new contents")
print(file.read())
print("Finished")
file.close()
```

Try It Yourself

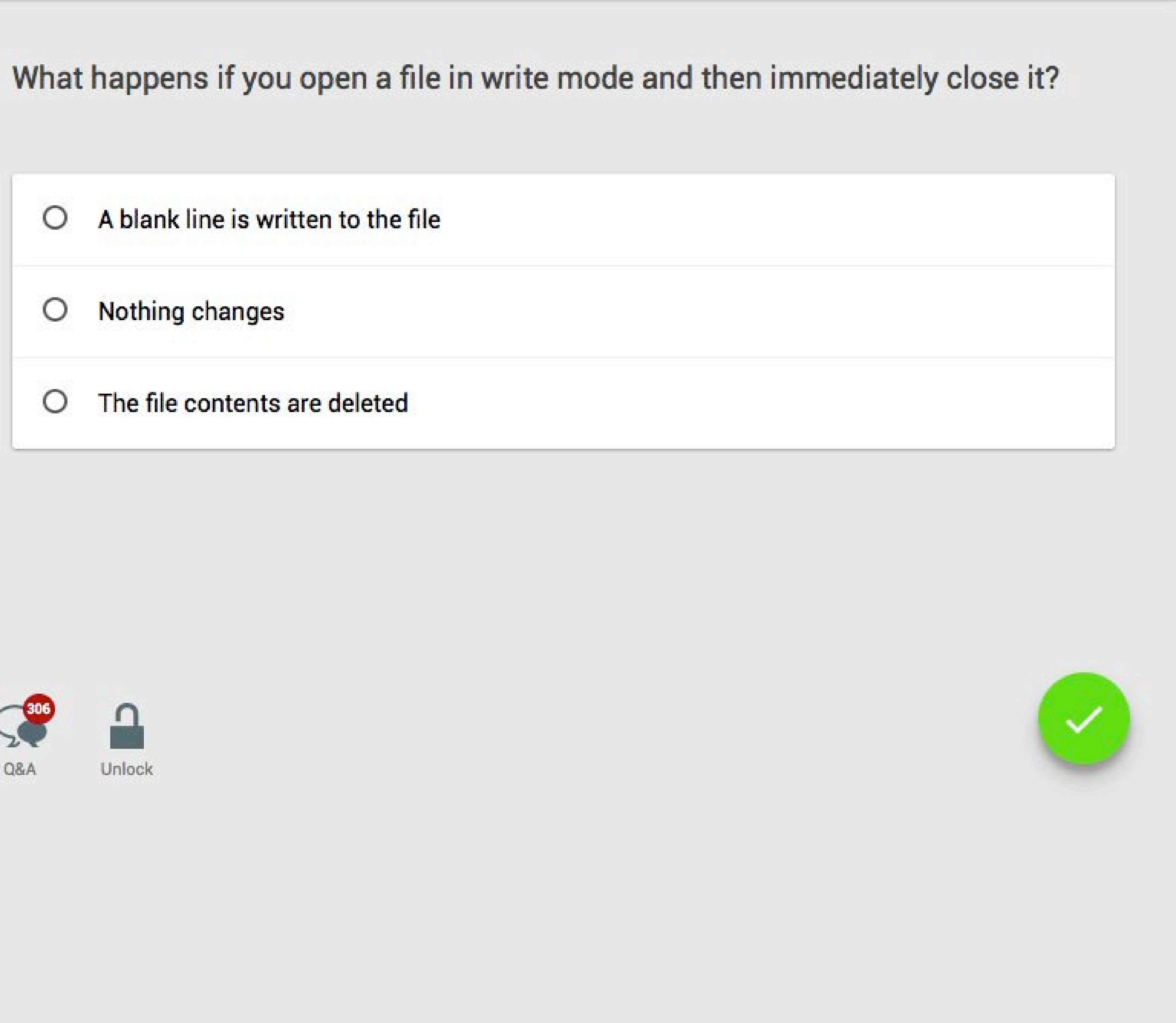
Result:

```
>>>
Reading initial contents
some initial text
Finished
Reading new contents
Some new text
Finished
>>>
```

As you can see, the content of the file has been overwritten.

117 COMMENTS

306
Q&A



What happens if you open a file in write mode and then immediately close it?

- ○ A blank line is written to the file
- ○ Nothing changes
- ○ The file contents are deleted

Writing Files

The **write** method returns the number of **bytes** written to a file, if successful.

```
msg = "Hello world!"
file = open("newfile.txt", "w")
amount_written = file.write(msg)
print(amount_written)
file.close()
```

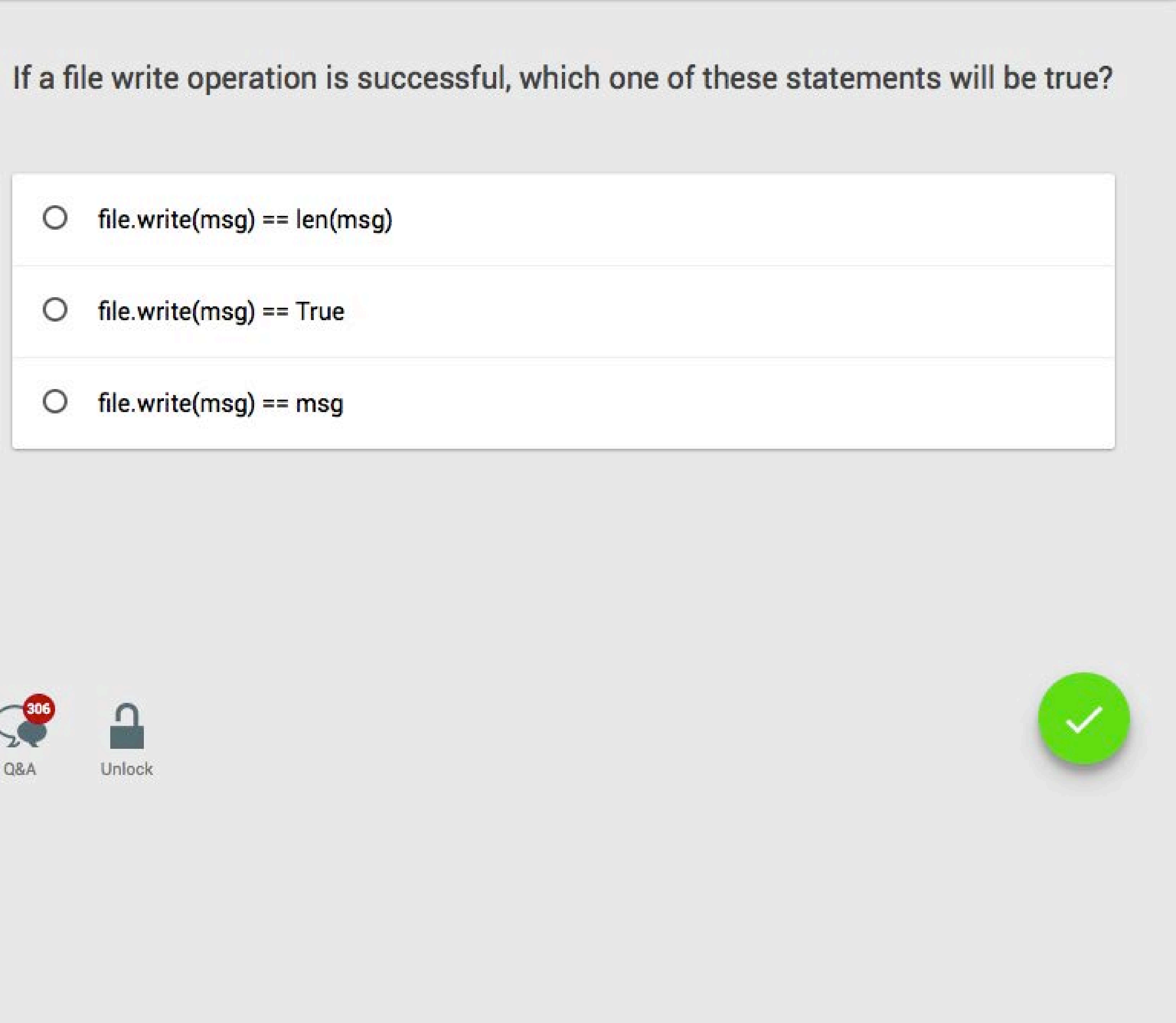
Try It Yourself

Result:

```
>>>
12
>>>
```

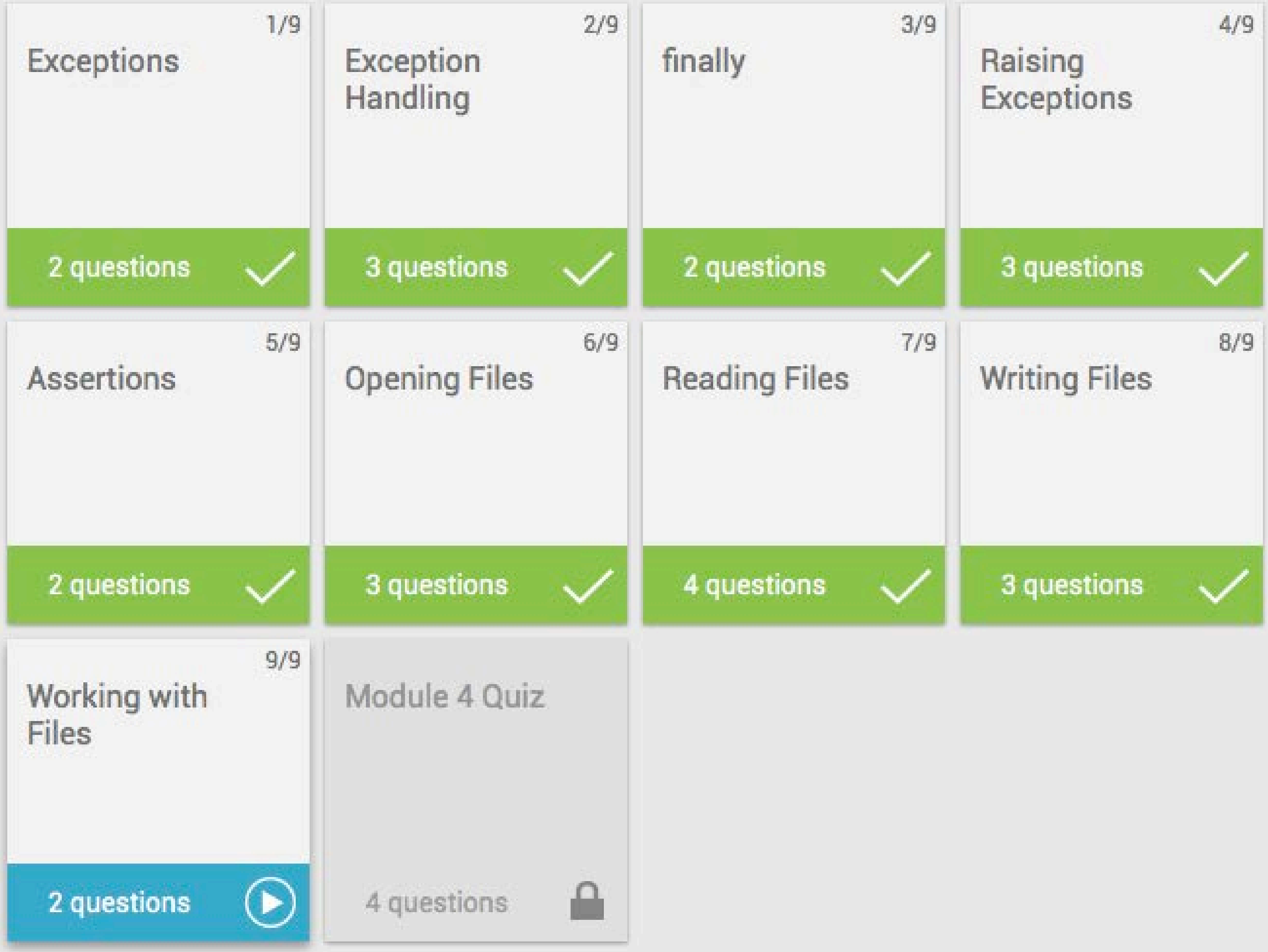
121 COMMENTS

306
Q&A



If a file write operation is successful, which one of these statements will be true?

- ○ file.write(msg) == len(msg)
- ○ file.write(msg) == True
- ○ file.write(msg) == msg



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions ✓

4/9
Raising Exceptions
3 questions ✓

5/9
Assertions
2 questions ✓

6/9
Opening Files
3 questions ✓

7/9
Reading Files
4 questions ✓

8/9
Writing Files
3 questions ✓

9/9
Working with Files
2 questions ▶

Module 4 Quiz
4 questions 🔒

Working with Files

It is good practice to avoid wasting resources by making sure that files are always closed after they have been used. One way of doing this is to use **try** and **finally**.

```
try:
  f = open("filename.txt")
  print(f.read())
finally:
  f.close()
```

This ensures that the file is always closed, even if an error occurs.

64 COMMENTS

306

Q&A

Will the close() function get called in this code?

```
try:
  f = open("filename.txt")
  print(f.read())
  print(1 / 0)
finally:
  f.close()
```

○ Yes

○ No

306 Q&A

Unlock

Working with Files

An alternative way of doing this is using **with** statements. This creates a temporary variable (often called **f**), which is only accessible in the indented block of the **with** statement.

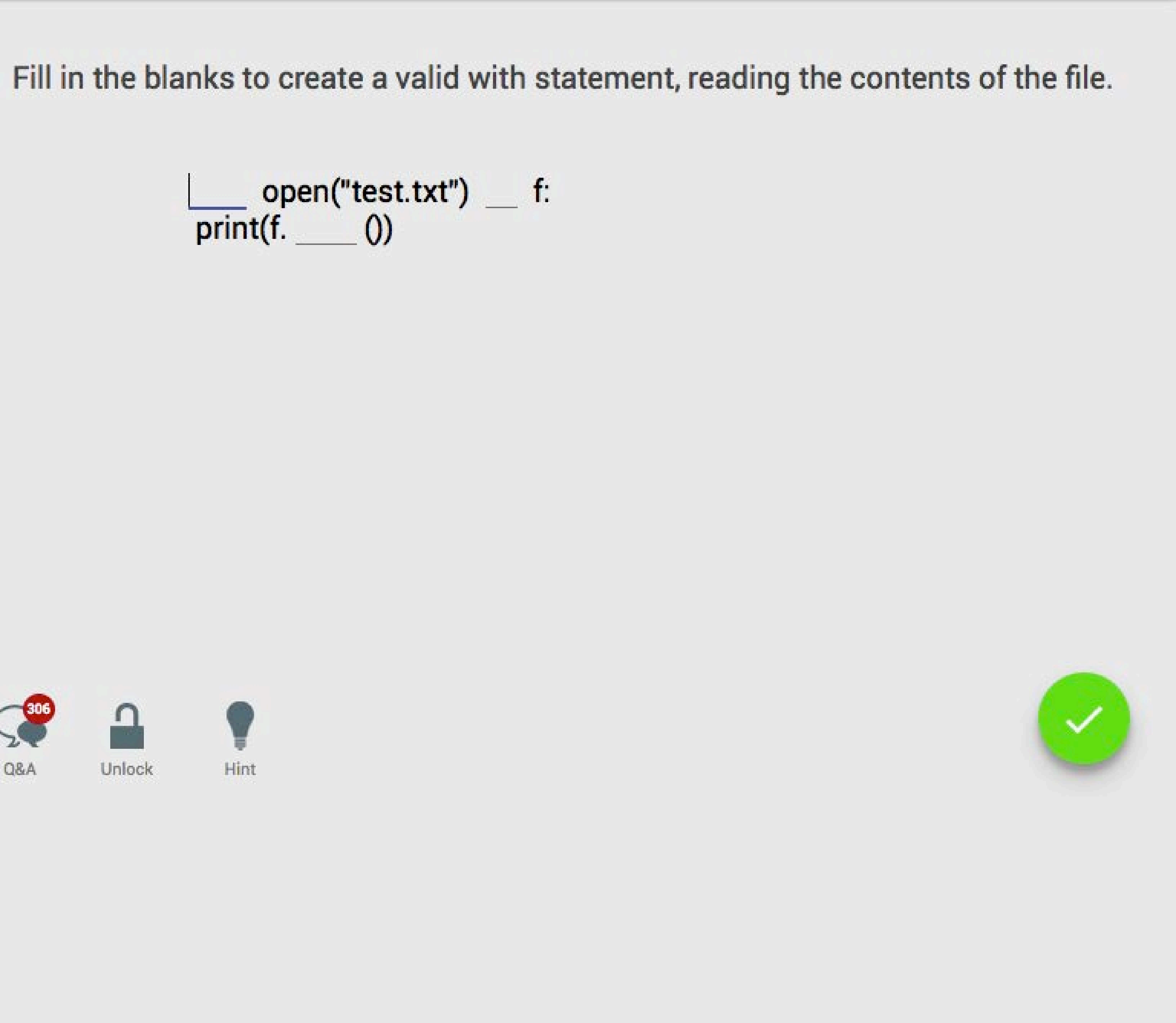
```
with open("filename.txt") as f:
  print(f.read())
```

The file is automatically closed at the end of the **with** statement, even if exceptions occur within it.

68 COMMENTS

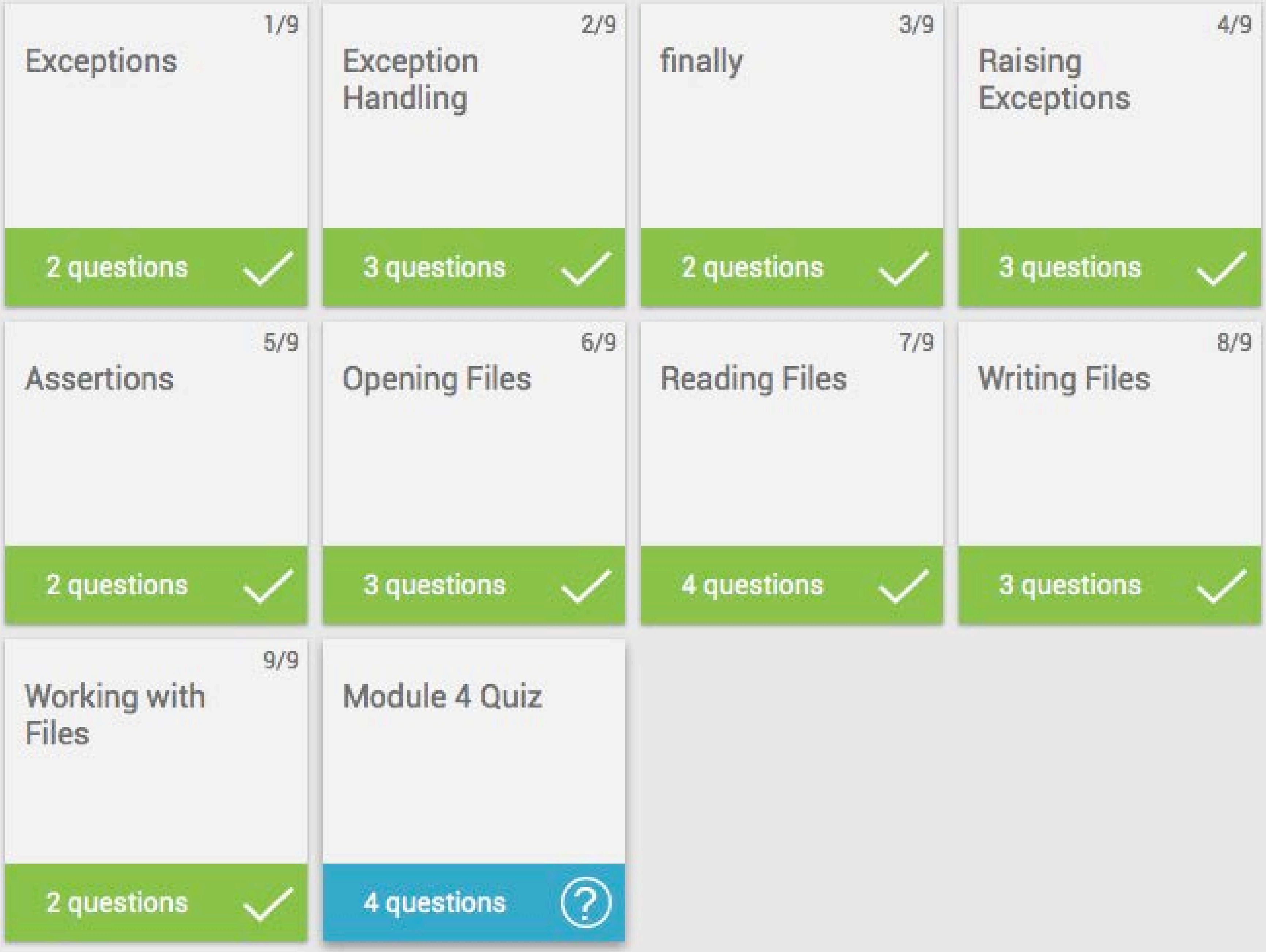
306

Q&A



Fill in the blanks to create a valid with statement, reading the contents of the file.

```
___ open("test.txt") __ f:
 print(f. ____ ())
```



1/9
Exceptions
2 questions ✓

2/9
Exception Handling
3 questions ✓

3/9
finally
2 questions ✓

4/9
Raising Exceptions
3 questions ✓

5/9
Assertions
2 questions ✓

6/9
Opening Files
3 questions ✓

7/9
Reading Files
4 questions ✓

8/9
Writing Files
3 questions ✓

9/9
Working with Files
2 questions ✓

Module 4 Quiz
4 questions ?

Which number is not printed by this code?

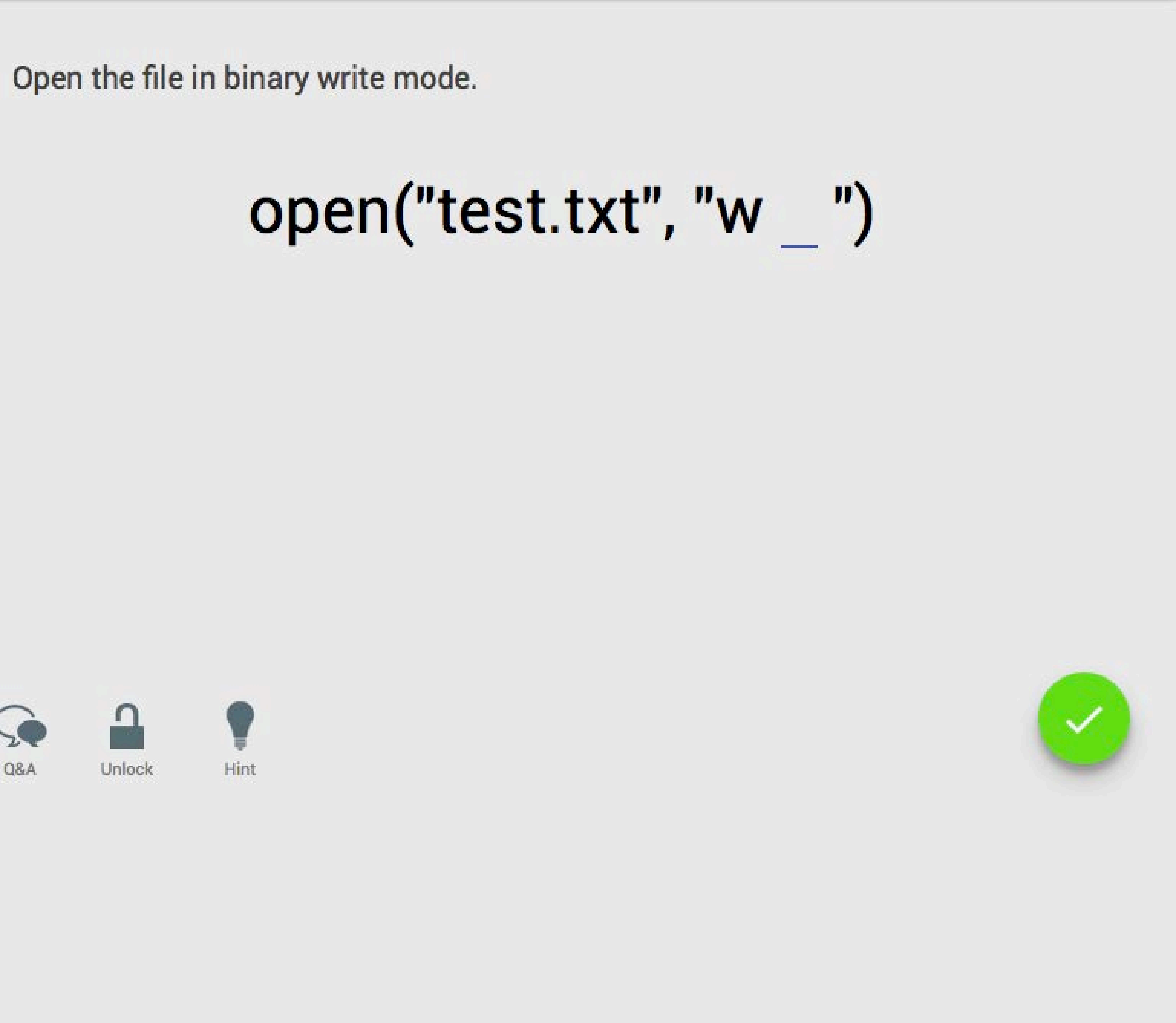
```
try:
  print(1)
  print(20 / 0)
  print(2)
except ZeroDivisionError:
  print(3)
finally:
  print(4)
```

○ 4

○ 3

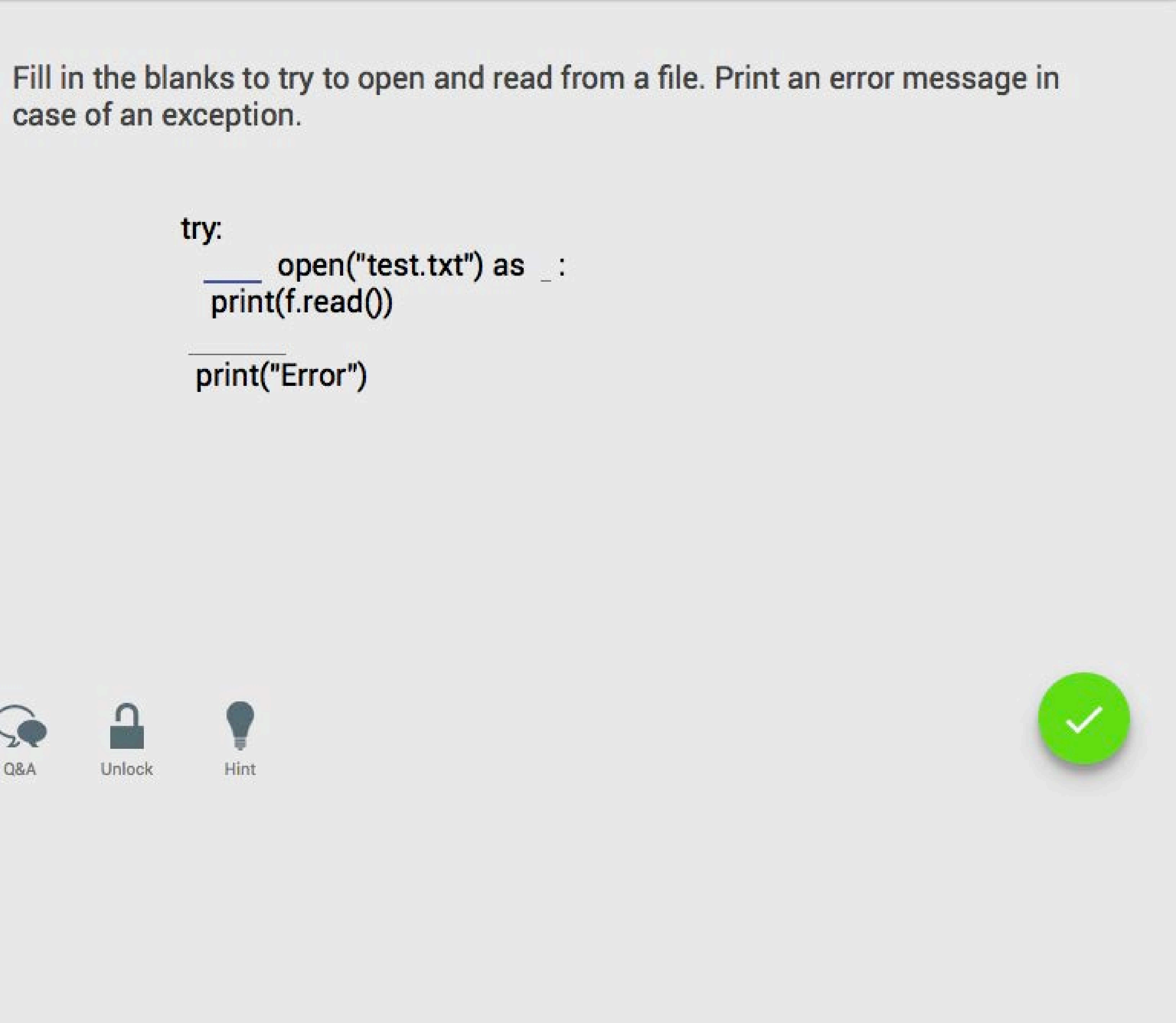
○ 2

Q&A Unlock



Open the file in binary write mode.

```
open("test.txt", "w _ ")
```



Fill in the blanks to try to open and read from a file. Print an error message in case of an exception.

```
try:
   ____ open("test.txt") as _:
    print(f.read())
  ________
  print("Error")
```

What is the highest number that would be printed by this code?

```
try:
  print(1)
  assert 2 + 2 == 5
except AssertionError:
  print(3)
except:
  print(4)
```

Q&A Unlock Hint