```
# print_lines.py
#
# ICS 32 Winter 2014
# Code Example
#
# This module implements a Python program that demonstrates an example of
# a function that can reasonably catch the exceptions it raises.



# print_number_of_lines() takes the path to a file, which is presumed
# to be a text file.  It determines how many lines of text are in the
# file and prints it to the console, while also printing an additional
# log of information about the process of determining it.  While the
# example is somewhat contrived -- in the sense that a much more useful
# function would be one that *only* returned the number of lines in the
# file, since it could be more flexibly combined with other functions
# -- it serves to demonstrate how a function catches exceptions.
#
# The control flow works like this:
#
# * The code in the "try" clause is attempted once
#
# * If it succeeds, the code in the "else" clause is executed, because
#   that code always executes when the "try" clause does not raise an
#   exception; also, the code in the "finally" clause is executed,
#   because that code is executed as we exit the entire "try" statement,
#   regardless of whether there was an exception.
#
# * If an exception is raised in the "try" clause, Python will compare
#   the kind of exception raised with those listed in each "except"
#   clause, then execute the code in the first "except" clause that
#   matches.  An "except" clause with no type listed (which always has
#   to be the last one, if there is one) catches any exception.  If no
#   clauses match -- impossible here, but possible if there is no "except"
#   clause without a type -- the exception is not caught, so it flows
#   back out to the function's caller.  Whether caught or not, the code in
#   the "finally" clause is executed regardless.
#
# Given all of that, take a look through the code here and try to decide,
# without running it,

def print_number_of_lines(path_to_file: str) -> None:
    '''
    Given the path to a file, prints the number of lines in that file to
    the console.
    '''

    # We begin by assigning f to None.  None is an object, so note that
    # this is different than not having assigned a value to f at all.
    # The problem is that the assignment to f in the "try" clause may
    # fail if the file can't be opened, in which case f will simply not
    # be bound at all; by assigning it None, we know that f will always
    # be bound no matter what, so our subsequent attempt in the "finally"
    # clause to check if opening succeeded will work.

    f = None

    # Now, we try actually opening the file and seeing how many lines of
    # text are in it.  The way we've done it here -- reading all lines
```

```
    # and then seeing how many we got -- has the nice property of being
    # short, though it is not actually a very good way to solve the
    # problem if the file might large, because this will load the entire
    # file into memory (at least temporarily).

    try:
        print('Opening the file')
        f = open(path_to_file)
        print('The file has {} lines of text'.format(len(f.readlines())))
    except FileNotFoundError:
        print('File not found')
    except OSError:
        print('The operating system reported a problem')
    except:
        print('I have no idea what happened, but it was not good')
    else:
        # This will be printed any time the entire operation succeeded
        # (i.e., no exceptions were raised).
        print('Succeeded')
    finally:
        # Here, we close the file, but only if it was already opened
        # successfully.  If not, the assignment of a file object to
        # f will have failed, so it will still have the value originally
        # assigned to it (None).
        if f != None:
            print('Closing the file')
            f.close()
        else:
            print("No need to close the file; it wasn't opened successfully")



def user_interface() -> None:
    '''
    Repeatedly asks the user to specify a file; each time, the number of
    lines of text in the file are printed, unless the file could not be
    opened, in which case a brief error message is displayed instead.
    '''

    while True:
        # Note the call to strip() here.  I'm taking the input we got back
        # from the user (a string) and stripping the spaces from either
        # end of it.
        path_to_file = input('What file? ').strip()

        if path_to_file == '':
            break
        else:
            print_number_of_lines(path_to_file)



# This "if" statement, where we check if __name__ is the string '__main__'
# is how a module can differentiate the scenario where it's being run
# directly from the scenario where it's been imported but not run.
# This makes our module a "program".

if __name__ == '__main__':
    user_interface()
```