```
# line_count.py
#
# ICS 32 Winter 2014
# Code Example
#
# This module demonstrates a more realistic design for a program similar
# to the one in print_lines.py.  The program counts the number of lines of
# text in a text file, by separating the program's functionality into
# self-contained parts:
#
# * A function that takes the path to a file and returns the number of
#   lines of text in it.
#
# * A function that acts as a user interface, both taking input from the
#   user and printing output.  All user interaction happens there.
#
# * A "main" block that makes the module executable as a program.
#
# It also improves the design in one other way: rather than reading the
# entire file into memory just so we can count the number of lines, we
# instead read the file one line at a time and count the lines as we go.
# Even though we've written more code, the program, overall, is probably
# doing roughly the same amount of work; it's just that we're doing a
# little more and the Python library is doing a little less.
#
# The change in our design also changes how exceptions are handled.
# Let's see how that turns out.



# count_lines_in_file() takes the path to a file and returns an
# integer specifiying the number of lines of text in that file.
#
# Notice that this function does not catch any exceptions, but that it
# still has a "try" statement with a "finally" clause.  That's because
# we're using the "finally" clause to ensure that the file is closed, if
# it was successfully opened, no matter what happens.  (For example,
# any of the calls to readline() could raise exceptions.  If so, the
# file will have been opened, so we'll want to ensure it gets closed.
# Or no exceptions might be raised, in which case we still want to be
# sure it gets closed.  A "finally" clause ensures both.)

def count_lines_in_file(path_to_file: str) -> int:
    '''
    Given the path to a file, returns the number of lines of text in
    that file, or raises an exception if the file could not be opened.
    '''

    f = None

    try:
        f = open(path_to_file, 'r')
        lines = 0
        line = f.readline()

        while line != '':
            lines += 1
            line = f.readline()

        return lines
```

```
    finally:
        if f != None:
            f.close()


# An interesting question to ask at this point is why count_lines_in_file
# doesn't catch exceptions, but instead steps aside and lets its caller
# handle them instead.
#
# Think about the function's job: it takes the path to a file and returns
# the number of lines of text in that file.  And here's the important
# thing: it can't possibly know where that path came from.  This function
# might be called by the user_interface() function below.  But it might also
# be called from the interpreter, or from code in another module.  There
# might have been a human user, but there might not.  This function's role
# is best kept simple, so it shouldn't make any assumptions about what its
# callers do.
#
# Given that, now we have to ask ourselves another question.  If this
# function doesn't assume anything about where its parameter came from,
# what can it possibly do if the parameter is the path to a file that
# doesn't exist or can't otherwise be opened?  It can't ask the user for
# another path, because there may not be a user.  It can't guess about
# what other file it might try, because there's no reasonable guess.
# All it can do is say "Well, I tried, but I failed!"  Failure to open
# the file is failure to count the number of lines in it, pure and
# simple.  In Python, that means it should step aside and let any
# exception propagate to its caller, who might be more aware of the
# broader context (e.g. is there a user?) and can do something appropriate.




# Our user_interface() function is different from the one in
# print_lines.py, in that it takes over the duty of printing the
# program's output, in addition to reading its input.
#
# Here, we're catching the exception raised in count_lines_in_file(),
# because this function is aware of the broader context.  There is a
# user and interaction is being done via the console.  So an appropriate
# thing to do might be to print an error message.

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:
        path_to_file = input('What file? ').strip()

        if path_to_file == '':
            break

        try:
            lines_in_file = count_lines_in_file(path_to_file)
            print('{} line(s) in {}'.format(lines_in_file, path_to_file))
        except:
            print('Failed')
```

```
# I should point out here that printing an error message to the console
# is not always what you do when you catch an exception, though it turned
# out to be reasonable enough in this example and print_lines.py.  We'll
# see plenty of examples where something else is more appropriate.



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