Practice Problem 7.2

For each name in the next module, indicate whether it is a global name or whether it is local in `f(x)` or local in `g(x)`.

Module: fandg.py

```
def f(y):
    x = 2
    return g(x)

def g(y):
    global x
    x = 4
    return x*y

x = 0
res = f(x)
print('x = {}, f(0) = {}'.format(x, res))
```

7.3 Exceptional Control Flow

While the focus of the discussion in this chapter has been on namespaces, we have also touched on another fundamental topic: how the operating system and namespaces support the "normal" execution control flow of a program, especially function calls. We consider, in this section, what happens when the "normal" execution control flow gets interrupted by an exception and ways to control this exceptional control flow. This section also continues the discussion of exceptions we started in Section 4.4.

Exceptions and Exceptional Control Flow

Error objects are called *exceptions* because when they are created, the normal execution flow of the program (as described by, say, the program's flowchart) is interrupted, and the execution switches to the so called *exceptional control flow* (which the flowchart typically does not show as it is not part of the normal program execution). The default exceptional control flow is to stop the program and print the error message contained in the exception object.

We illustrate this using the functions `f()`, `g()`, and `h()` we defined in Section 7.1. In Figure 7.2, we illustrated the normal flow of execution of function call `f(4)`. In Figure 7.10, we illustrate what happens when we make the function call `f(2)` from the shell.

The execution runs normally all the way to function call `h(0)`. During the execution of `h(0)`, the value of `n` is 0. Therefore, an error state occurs when the expression `1/n` is evaluated. The interpreter raises a `ZeroDivisionError` exception and creates a `ZeroDivisionError` exception object that contains information about the error.

The default behavior when an exception is raised is to interrupt the function call in which the error occurred. Because the error occurred while executing `h(0)`, the execution of `h(0)` is interrupted. However, the error also occurred during the execution of function calls `g(1)` and `f(2)`, and the execution of both is interrupted as well. Thus, statements shown in gray in Figure 7.10 are never executed.

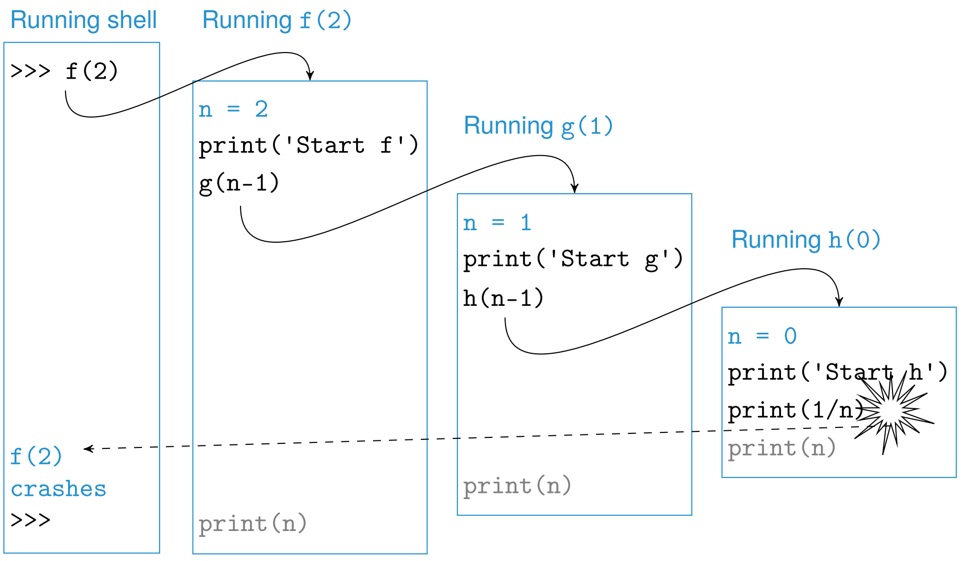

Figure 7.10 Execution of `f(2)`**.** The normal execution control flow of function call `f(2)` from the shell is shown with black arrows: `f(2)` calls `g(1)`, which, in turn, calla `h(0)`. When the evaluation of expression `1/n` = 1/0 is attempted, a `ZeroDivisionError` exception is raised. The normal execution control flow is interrupted: Function call `h(0)` does not run to completion, and neither do `g(1)` or `f(2)`. The exceptional control flow is shown with a dashed arrow. Statements that are not executed are shown in gray. Since call `f(2)` is interrupted, the error information is output in the shell.

When execution returns to the shell, the information contained in the exception object is printed in the shell:

```
Traceback (most recent call last):
  File "<pyshell#116>", line 1, in <module>
    f(2)
  File "/Users/me/ch7.py", line 13, in f
    g(n-1)
  File "/Users/me/ch7.py", line 8, in g
    h(n-1)
  File "/Users/me/ch7.py", line 3, in h
    print(1/n)
ZeroDivisionError: division by zero
```

In addition to the type of error and a friendly error message, the output also includes a *traceback*, which consists of all the function calls that got interrupted by the error.

Catching and Handling Exceptions

Some programs should not terminate when an exception is raised: server programs, shell programs, and pretty much any program that handles requests. Since these programs receive requests from outside the program (interactively from the user or from a file), it is difficult to ensure that the program will not enter an erroneous state because of malformed input. These programs need to continue providing their service even if internal errors occur. What this means is that the default behavior of stopping the program when an error occurs and printing an error message must be changed.

We can change the default exceptional control flow by specifying an alternate behavior when an exception is raised. We do this using the `try/except` pair of statements. The next small application illustrates how to use them:

Module: age1.py

```python
strAge = input('Enter your age: ')
intAge = int(strAge)
print('You are {} years old.'.format(intAge))
```

The application asks the user to interactively enter her age. The value entered by the user is a string. This value is converted to an integer before being printed. Try it!

This program works fine as long as the user enters her age in a way that makes the conversion to an integer possible. But what if the user types “fifteen” instead?

```
>>>
Enter your age: fifteen
Traceback (most recent call last):
  File "/Users/me/age1.py", line 2, in <module>
    intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'
```

A `ValueError` exception is raised because string `'fifteen'` cannot be converted to an integer.

Instead of “crashing” while executing the statement `age = int(strAge)`, wouldn’t it be nicer if we could tell users that they were supposed to enter their age using decimal digits. We can achieve this using the next `try` and `except` pair of statements:

Module: age2.py

```python
1  try:
2      # try block --- executed first; if an exception is
3      # raised, the execution of the try block is interrupted
4      strAge = input('Enter your age: ')
5      intAge = int(strAge)
6      print('You are {} years old.'.format(intAge))
7  except:
8      # except block --- executed only if an exception
9      # is raised while executing the try block
10     print('Enter your age using digits 0-9!')
```

The `try` and `except` statements work in tandem. Each has an indented code block below it. The code block below the `try` statement, from line 2 to line 6, is executed first. If no errors occur, then the code block below `except` is ignored:

```
>>>
Enter your age: 22
You are 22 years old.
```

If, however, an exception is raised during the execution of a `try` code block (say, `strAge` cannot be converted to an integer), the Python interpreter will skip the execution of the remaining statements in the `try` code block and execute the code block of the `except` statement (i.e., line 9) instead:

```
>>>
Enter your age: fifteen
Enter your age using digits 0-9!
```

Note that the first line of the `try` block got executed but not the last.

The format of a `try/except` pair of statements is:

```
try:
    <indented code block 1>
except:
    <indented code block 2>
<non-indented statement>
```

The execution of `<indented code block 1>` is attempted first. If it goes through without any raised exceptions, then `<indented code block 2>` is ignored and execution continues with `<non-indented statement>`. If, however, an exception is raised during the execution of `<indented code block 1>`, then the remaining statements in `<indented code block 1>` are not executed; instead `<indented code block 2>` is executed. If `<indented code block 2>` runs to completion without a new exception being raised, then the execution continues with `<non-indented statement>`.

The code block `<indented code block 2>` is referred to as the *exception handler* because it handles a raised exception. We will also say that an `except` statement *catches* an exception.

The Default Exception Handler

If a raised exception is not caught by an `except` statement (and thus not handled by a user-defined exception handler), the executing program will be interrupted and the traceback and information about the error are output. We saw this behavior when we ran module `age1.py` and entered the age as a string:

```
>>>
Enter your age: fifteen
Traceback (most recent call last):
  File "/Users/me/age1.py", line 2, in <module>
    intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'
```

This default behavior is actually the work of Python's *default exception handler*. In other words, every raised exception will be caught and handled, if not by a user-defined handler then by the default exception handler.

Catching Exceptions of a Given Type

In the module `age2.py`, the `except` statement can catch an exception of any type. The `except` statement could also be written to catch only a certain type of exception, say `ValueError` exceptions:

Module: age3.py

```
try:
    # try block
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'.format(intAge))
except ValueError:
    # except block --- executed only if a ValueError
    # exception is raised in the try block
    print('Enter your age using digits 0-9!')
```

If an exception is raised while executing the try code block, then the exception handler is executed only if the type of the exception object matches the exception type specified in the corresponding except statement (ValueError in this case). If an exception is raised that does match the type specified in the except statement, then the except statement will not catch it. Instead, the default exception handler will handle it.

Multiple Exception Handlers

There could be not just one but several except statements following one try statement, each with its own exception handler. We illustrate this with the next function readAge(), which attempts to open a file, read the first line, and convert it to an integer in a single try code block.

Module: ch7.py

```
def readAge(filename):
    '''converts first line of file filename to
       an integer and prints it'''
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is', age)
    except IOError:
        # executed only if an IOError exception is raised
        print('Input/Output error.')
    except ValueError:
        # executed only if a ValueError exception is raised
        print('Value cannot be converted to integer.')
    except:
        # executed if an exception other than IOError
        # or ValueError is raised
        print('Other error.')
```

Several types of exceptions could be raised while executing the try code block in function readAge. The file might not exist:

```
>>> readAge('agg.txt')
Input/Output error.
```

In this case, what happened was that an IOError exception got raised while executing the first statement of the try code block; the remaining statements in the code section were skipped and the IOError exception handler got executed.

Another error could be that the first line of the file age.txt does not contain something that can be converted to an integer value:

File: age.txt

```
>>> readAge('age.txt')
Value cannot be converted to integer
```

The first line of file age.txt is 'fifteen\n', so a ValueError exception is raised when attempting to convert it to an integer. The associated exception handler prints the friendly message without interrupting the program.

The last except statement will catch any exception that the first two except statements did not catch.