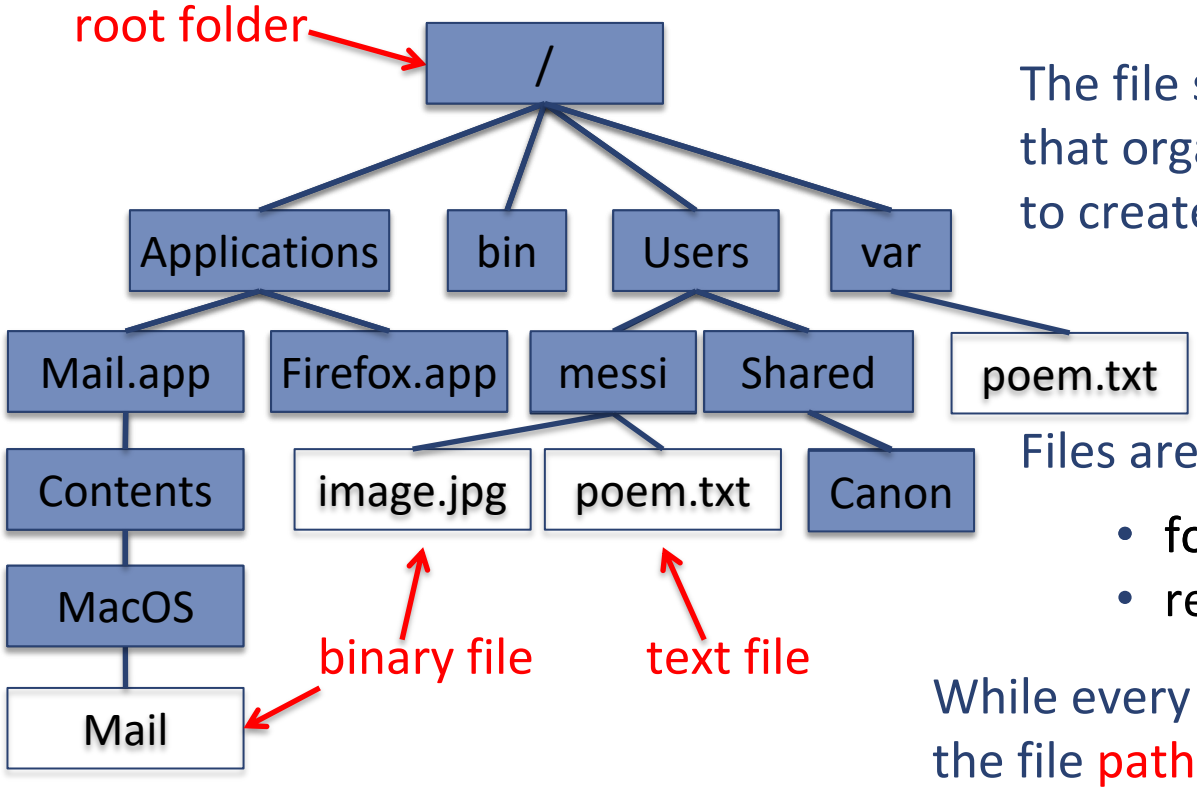
Text Data, File I/O, and Exceptions

- File Input/Output
- Errors and Exceptions

Files and the file system



The file system is the OS component that organizes files and provides a way to create, access, and modify files

Files are organized into a tree structure

- folders (or directories)
- regular files

While every file and folder has a name, it is the file pathname that identifies the file

Absolute pathnames

- `/var/poem.txt`
- `/Users/messi/poem.txt`
- `/Applications/Mail.app/`

Relative pathnames (relative to current working directory `Users`)

- `messi/poem.txt`
- `messi/image.jpg`
- `Shared/`

Opening and closing a file

Processing a file consists of:

1. Opening the file
2. Reading from and/or writing to the file
3. Closing the file

Built-in function `open()` is used to open a file

- The first input argument is the file pathname, whether absolute or relative with respect to the current working directory
- The second (optional) argument is the file mode
- Returns a "file" object

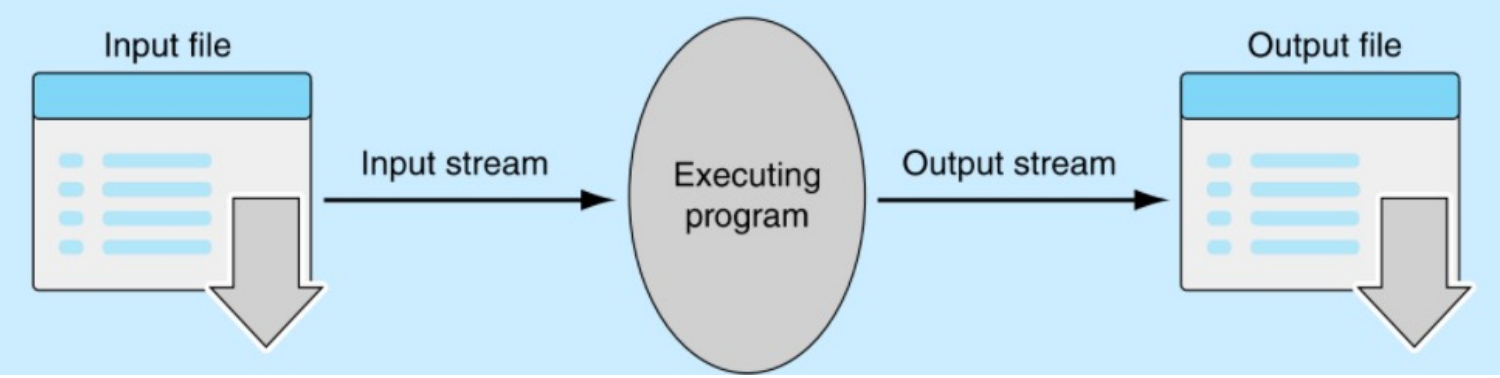
File mode `'r'` is used to open a file for reading (rather than, say, writing)

```
>>> infile = open('sample.txt')
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    infile = open('sample.txt')
IOError: [Errno 2] No such file or directory:
'sample.txt'
>>> infile = open('example.txt', 'r')
>>> infile.close()
>>>
```

A "file" object is of a type that supports several "file" methods, including method `close()` that closes the file

More on File Object

- When opening a file, we create a **file object** or **file stream** that is a connection between the file information on disk and the program
- The stream contains a **buffer** of the information from the file and make it accessible to the program



Taken from "The Practice of Computing using Python" by Punch and Enbody, 2013

Buffering

- Reading from a disk is very slow. Thus the computer will read a lot of data from a file in the hopes that, if you need the data in the future, it will be buffered in the file object.
- This means that the file object contains a copy of information from the file called a **cache** (pronounced "cash")

Open file mode

The file mode defines how the file will be accessed

Mode	Description
`r`	Reading (default)
`w`	Writing (if file exists, content is wiped)
`a`	Append (if file exists, writes are appended)
`r+`	Reading and Writing
`t`	Text (default)
`b`	Binary

These are all equivalent →

```
>>> infile = open('example.txt', 'rt')
>>> infile = open('example.txt', 'r')
>>> infile = open('example.txt', 't')
>>> infile = open('example.txt')
```

File methods

There are several "file" types; they all support similar "file" methods

- Methods `read()` and `readline()` return the characters read as a string
- Methods `readlines()` returns the characters read as a list of lines
- Method `write()` returns the number of characters written

Usage	Description
`infile.read(n)`	Read `n` characters starting from cursor; if fewer than `n` characters remain, read until the end of file
`infile.read()`	Read starting from cursor up to the end of the file
`infile.readline()`	Read starting from cursor up to, and including, the end of line character
`infile.readlines()`	Read starting from cursor up to the end of the file and return list of lines
`outfile.write(s)`	Write string `s` to file `outfile` starting from cursor
`infile.close(n)`	Close file `infile`

Reading a file

```
1 The 3 lines in this file end with the new line character.\n
2 \n
3 There is a blank line above this line.\n
```

example.txt

When the file is opened, a cursor is associated with the opened file

The initial position of the cursor is:

- at the beginning of the file, if file mode is `r`
- at the end of the file, if file mode is `a` or `w`

```
>>> infile = open('example.txt')
>>> infile.read(1)
'T'
>>> infile.read(5)
'he 3 '
>>> infile.readline()
'lines in this file end with the new line
character.\n'
>>> infile.read()
'\nThere is a blank line above this line.\n'
>>> infile.close()
>>>
```

Patterns for reading a text file

Common patterns for reading a file:

1. Read the file content into a string
2. Read the file content into a list of words
3. Read the file content into a list of lines

Example:

```
def num_chars(filename):
    'returns the number of characters in file filename'

    infile = open(filename, 'r')
    content = infile.read()
    infile.close()

    return len(content)
```

Patterns for reading a text file

Common patterns for reading a file:

1. Read the file content into a string
2. Read the file content into a list of words
3. Read the file content into a list of lines

Example:

```
def num_words(filename):
    'returns the number of words in file filename'

    infile = open(filename)
    content = infile.read()
    infile.close()
    wordList = content.split()

    return len(wordList)
```

Patterns for reading a text file

Common patterns for reading a file:

1. Read the file content into a string
2. Read the file content into a list of words
3. Read the file content into a list of lines

Example:

```
def num_lines(filename):
    'returns the number of lines in file filename'

    infile = open(filename, 'r')
    lineList = infile.readlines()
    infile.close()

    return len(lineList)
```

Writing to a text file

```
1 This is the first line. Still the first line…\n
2 Now we are in the second line.\n
3 Non string value like 5 must be converted first.\n
4 Non string value like 5 must be converted first.\n
```

test.txt

```
>>> outfile = open('test.txt', 'w')
>>> outfile.write('T')
1
>>> outfile.write('his is the first line.')
22
>>> outfile.write(' Still the first line...\n')
25
>>> outfile.write('Now we are in the second line.\n')
31
>>> outfile.write('Non string value like '+str(5)+' must be converted first.\n')
49
>>> outfile.write('Non string value like {} must be converted first.\n'.format(5))
49
>>> outfile.close()
```

Types of errors

We saw different types of errors in this chapter

There are basically two types of errors:

- syntax errors
- runtime errors
- Semantic/logic errors

```
>>> excuse = 'I'm sick'
SyntaxError: invalid syntax
>>> print(hour+':'+minute+':'+second)
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    print(hour+':'+minute+':'+second)
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>> infile = open('sample.txt')
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    infile = open('sample.txt')
IOError: [Errno 2] No such file or directory:
'sample.txt'
```

Syntax errors

Syntax errors are errors that are due to the incorrect format of a Python statement

- They occur while the statement is being translated to machine language and before it is being executed.

```
>>> (3+4]
SyntaxError: invalid syntax
>>> if x == 5
SyntaxError: invalid syntax
>>> print 'hello'
SyntaxError: invalid syntax
>>> lst = [4;5;6]
SyntaxError: invalid syntax
>>> for i in range(10):
print(i)
SyntaxError: expected an indented block
```

Runtime errors

The program execution gets into an erroneous state

When an error occurs, an "error" object is created

- This object has a type that is related to the *type* of error
- The object contains information about the error
- The default behavior is to print this information and interrupt the execution of the statement.

The "error" object is called an exception; the creation of an exception due to an error is called the raising of an exception

```
>>> 8/0
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    8/0
ZeroDivisionError: integer division or modulo by zero
>>> lst
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    lst
NameError: name 'lst' is not defined
>>> lst = [12, 13, 14]
>>> lst[3]
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    lst[3]
IndexError: list index out of range
>>> lst * lst
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    lst * lst
TypeError: can't multiply sequence by non-int of type 'list'
>>> int('4.5')
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    int('4.5')
ValueError: invalid literal for int() with base 10: '4.5'
```

Exception types

Some of the built-in exception classes:

Exception	Explanation
`KeyboardInterrupt`	Raised when user hits Ctrl-C, the interrupt key
`OverflowError`	Raised when a floating-point expression evaluates to a value that is too large
`ZeroDivisionError`	Raised when attempting to divide by 0
`IOError`	Raised when an I/O operation fails for an I/O-related reason
`IndexError`	Raised when a sequence index is outside the range of valid indexes
`NameError`	Raised when attempting to evaluate an unassigned identifier (name)
`TypeError`	Raised when an operation of function is applied to an object of the wrong type
`ValueError`	Raised when operation or function has an argument of the right type but incorrect value

Exceptions, revisited

Recall that when the program execution gets into an erroneous state, an exception object is created

- This object has a type that is related to the *type* of error
- The object contains information about the error
- The default behavior is to print this information and interrupt the execution of the statement that "caused" the error

The reason behind the term "exception" is that when an error occurs and an exception object is created, the normal execution flow of the program is interrupted and execution switches to the exceptional control flow

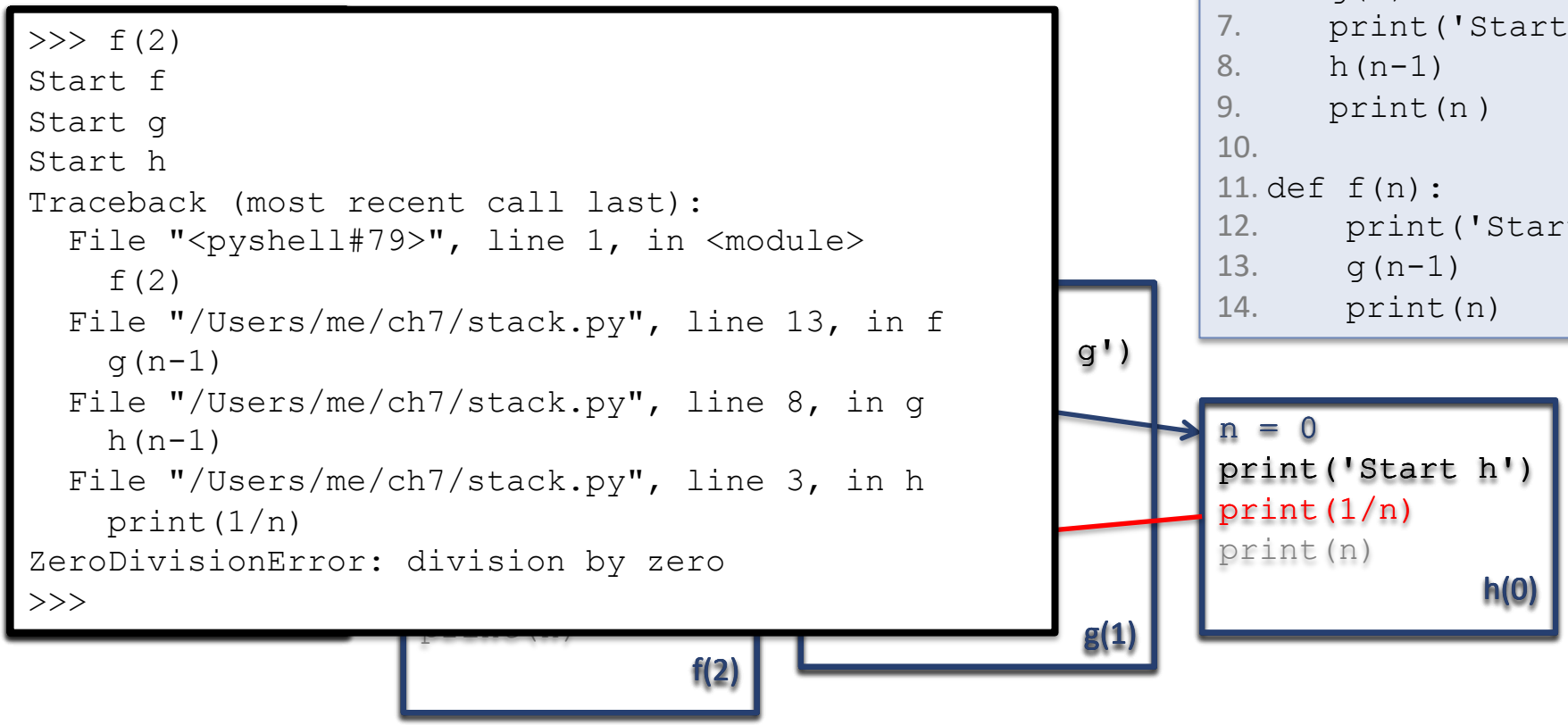
Exceptional control flow

Exceptional control flow

The default behavior is to interrupt the execution of each "active" statement and print the error information contained in the exception object.

```
>>> f(2)
Start f
Start g
Start h
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    f(2)
  File "/Users/me/ch7/stack.py", line 13, in f
    g(n-1)
  File "/Users/me/ch7/stack.py", line 8, in g
    h(n-1)
  File "/Users/me/ch7/stack.py", line 3, in h
    print(1/n)
ZeroDivisionError: division by zero
>>>
```

```
1. def h(n):
2.     print('Start h')
3.     print(1/n)
4.     print(n)
5.
6. def g(n):
7.     print('Start g')
8.     h(n-1)
9.     print(n)
10.
11. def f(n):
12.     print('Start f')
13.     g(n-1)
14.     print(n)
```



Catching and handling exceptions

It is possible to override the default behavior (print error information and "crash") when an exception is raised, using `try/except` statements

If an exception is raised while executing the `try` block, then the block of the associated `except` statement is executed

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'.format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

The `except` code block is the exception handler

Default behavior:

Custom behavior:

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

```
    intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'
>>>
```

Exception handling is not a conditional statements

- We perform handling exception to catch and handle **error that may occurred** in our program. Meaning that we **don't expect it during normal flow**
- We perform **conditional statements** to provide **multiple flow depending on the context during normal flow**
- This means **`if-else` statement** is not a correct way to perform exception handling

Further reading material:

- https://docs.python.org/3/glossary.html#term-lbyl
- https://docs.python.org/3/glossary.html#term-eafp

Format of a `try/except` statement pair

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

```
try:
    <indented code block>
except:
    <exception handler block>
<non-indented statement>
```

The exception handler handles any exception raised in the `try` block

The `except` statement is said to catch the (raised) exception

It is possible to restrict the `except` statement to catch exceptions of a specific type only

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
<non-indented statement>
```

Format of a `try/except` statement pair

```
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 ValueError:
        print('Value cannot be converted to integer.')
```

It is possible to restrict the `except` statement to catch exceptions of a specific type only

```
1 fifteen
```

age.txt

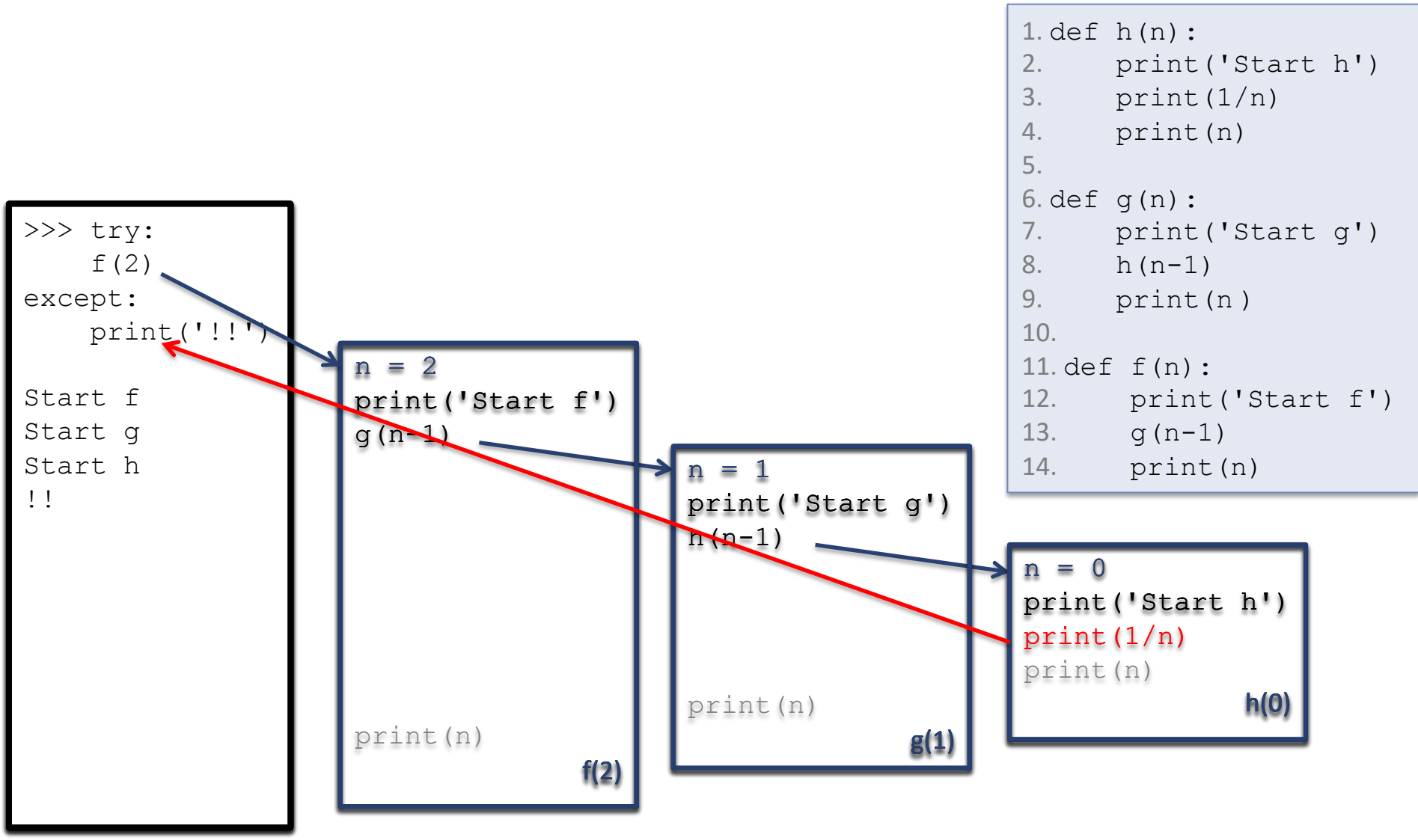
default exception handler prints this

```
>>> readAge('age.txt')
Value cannot be converted to integer.
>>> readAge('age.text')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    readAge('age.text')
  File "/Users/me/ch7.py", line 12, in readAge
    infile = open(filename)
IOError: [Errno 2] No such file or directory: 'age.text'
>>>
```

Multiple exception handlers

```
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.')
```

Controlling the exceptional control flow



Would you like to know more?

ADVANCED TOPIC

(Adv.) Other Files and Folders Processing…

Method	Description
os.remove(path_to_file)	Remove file pointed by **path_to_file**
os.rename(curr_filename, new_filename)	Change the name of file from **curr_filename** to **new_filename**
os.mkdir(path_to_folder)	Create a folder pointed by **path_to_folder**
os.getcwd()	Get **current working directory**

Using `with` keyword

```
''' 1) without using with statement
file = open('file_path', 'w')
file.write('hello world !')
file.close()
'''

'''
2) without using with statement
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()
'''

with open('dudu.txt', 'w+') as file:
    file.write('hello world')
```

Practice makes Perfect

HOMEWORK

Using Think Computer Science Book

- Open Exercise 13.11, please work on number 1 and 2.
- (Optional) Open Exercise 13.11, please work on number 3 and 4.