Introduction to Programming

CSC 401

Week 8

FUNCTIONS

Why use functions

- Benefits of functions
  - Code reuse
    - A fragment of code that is used multiple times in a program
  - Modularity
    - Reduce large program complexity; ease programming and testing breaking down the program into smaller, simpler, self-contained pieces
  - Encapsulation
    - An information-hiding mechanism

Encapsulation in functions

Main code

```
import random
x = random.randrange(1,9)
```

Encapsulated function randrange

Encapsulation through local variables

IDLE shell

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

Before executing function `double()`, variables `x` and `y` do not exist

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

After executing function `double()`, variables `x` and `y` still do not exist

`x` and `y` exist only during the execution of function call `double(5)`;

they are said to be local variables of function `double()`

Function call namespace

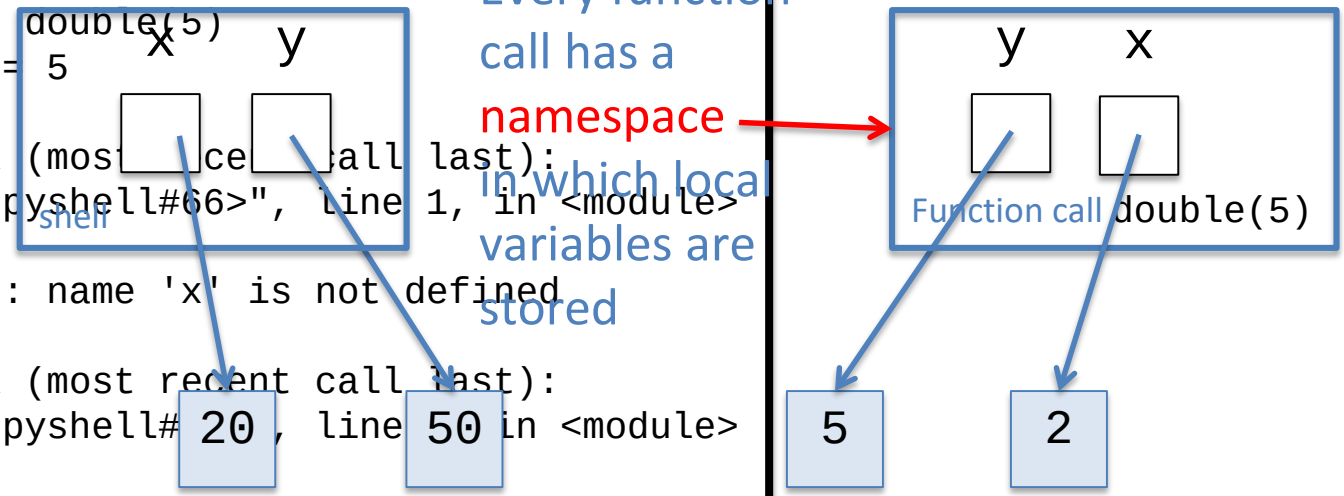
IDLE shell

```
>>> x, y = 20, 50
>>> res = double(5)
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

How is it possible that the values of x and y do not interfere with each other?

Even during the execution of `double()`, local variables `x` and `y` are invisible outside of the function!

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```



Every function call has a namespace in which local variables are stored

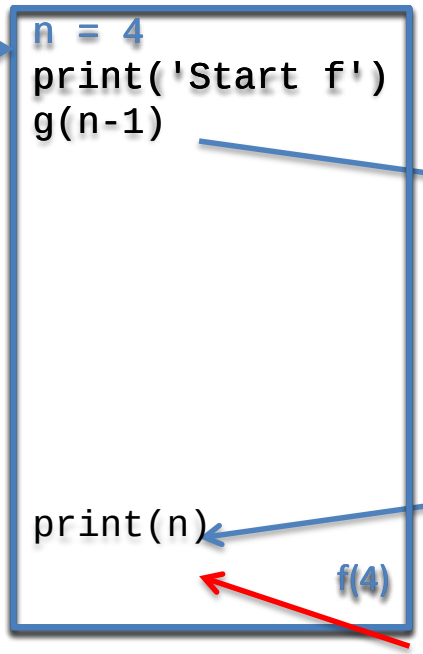
Function call namespace

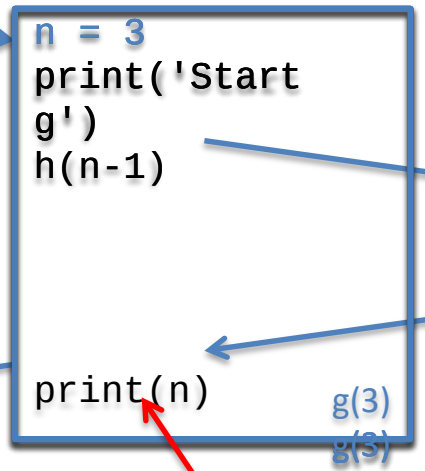
Every function call has a namespace in which local variables are stored

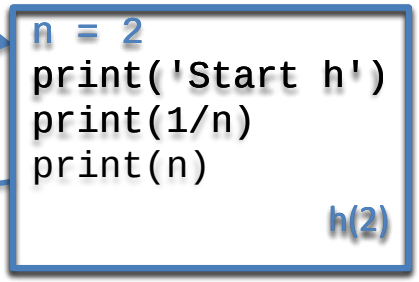
Note that there are several active values of n, one in each namespace; how are all the namespaces managed by Python?

```
def h(n):
    print('Start h')
    print(1/n)
    print(n)

def g(n):
    print('Start g')
    h(n-1)
    print(n)

def f(n):
    print('Start f')
    g(n-1)
    print(n)
```

```
>>> f(4)
Start f
Start g
Start h
0.5
2
3
4
```





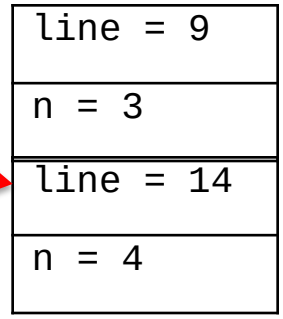


How does Python know which line to return to?

Program stack

The system dedicates a chunk of memory to the program stack; its job is to remember the values defined in a function call and …

… the statement to be executed after g(n-1) returns



Program stack

```
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)
```

```
>>> f(4)
Start f
Start g
Start h
0.5
2
3
4
```

```
n = 4
print('Start f')
g(n-1)

print(n)
```
f(4)

```
n = 3
print('Start g')
h(n-1)

print(n)
```
g(3)

```
n = 2
print('Start h')
print(1/n)
print(n)
```
h(2)

Scope and global vs. local namespace

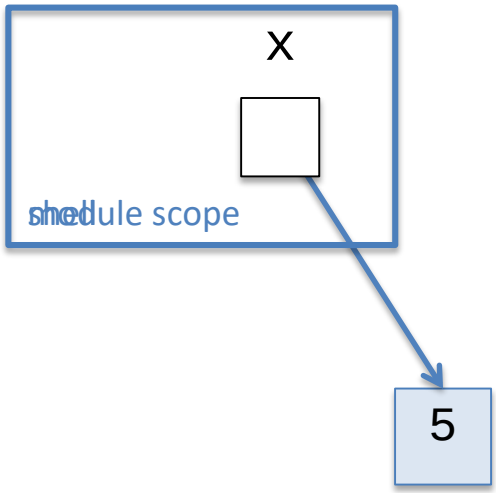
Every function call has a namespace associated with it.

- This namespace is where names defined during the execution of the function (e.g., local variables) live.
- The scope of these names (i.e., the space where they live) is the namespace of the function.

In fact, every name in a Python program has a scope

- Whether the name is of a variable, function, class, …
- Outside of its scope, the name does not exist, and any reference to it will result in an error.
- Names assigned/defined in the interpreter shell or in a module and outside of any function are said to have global scope.

Scope and global vs. local namespace




```
x = 5
```

scope.py

```
>>> x = 5
>>> x
5
>>>
```

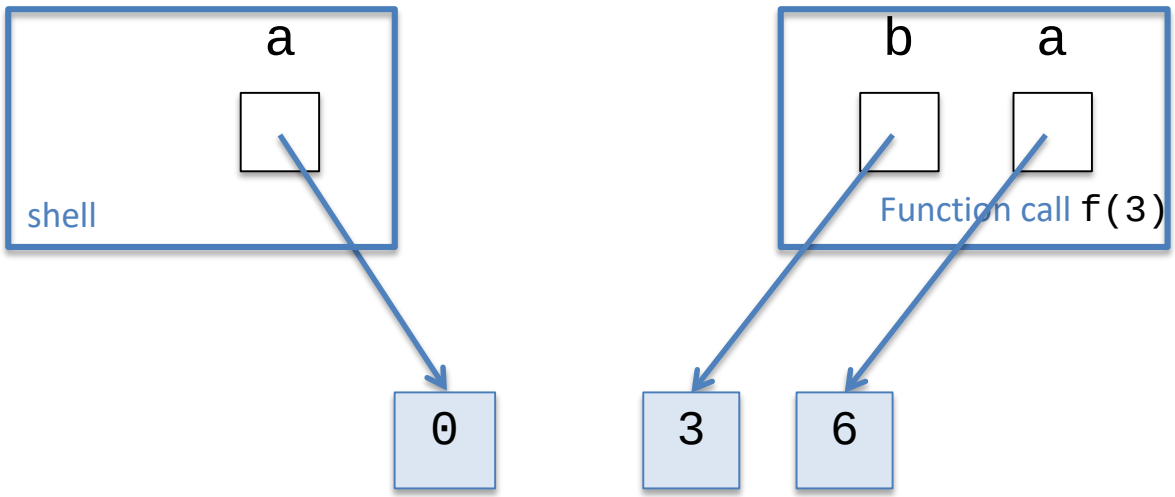
In fact, every name in a Python program has a scope

- Whether the name is of a variable, function, class, …
- Outside of its scope, the name does not exist, and any reference to it will result in an error.
- Names assigned/defined in the interpreter shell or in a module and outside of any function are said to have global scope. Their scope is the namespace associated with the shell or the whole module. Variables with global scope are referred to as global variables.

Example: variable with local scope

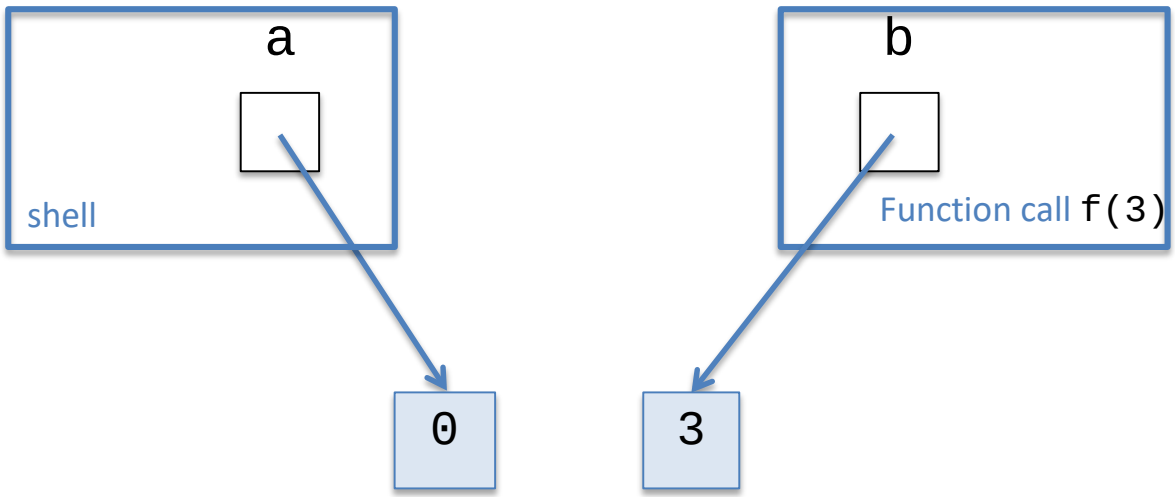
```
def f(b):        # f has global scope, b has local scope
    a = 6        # this a has scope local to function call f()
    return a*b   # this a is the local a

a = 0            # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```

```
>>> === RESTART ====
>>>
f(3) = 18
a is 0
>>>
```



Example: variable with global scope

```
def f(b):       # f has global scope, b has local scope
    return a*b  # this a is the global a

a = 0           # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```




```
>>> === RESTART ====
>>>
f(3) = 0
a is 0
>>>
```

Exercise

- Without using IDLE predict the output of the code below (Local1.py)

```
def myFunction(c):
    a=2
    b=3
    return(a*b + c)

a = 7
b = 5
c= myFunction(a)
print(a,b,c)
```

Exercise

- Without using IDLE predict the output of the code below (Local2.py)

```
def myFunction2(c):
    print(a)
    b=3
    return(a*b + c)

a = 7
b = 5
c= myFunction2(a)
print(a,b,c)
```

Exercise

- Without using IDLE predict the output of the code below (Local3.py)

```
def myFunction3(c):
    print(a)
    a=2
    b=3
    return(a*b + c)

a = 7
b = 5
c= myFunction3(a)
print(a,b,c)
```

Exercise

- Without using IDLE predict the output of the fragment of code below (Local4.py)

```
def myFunction4():
    return(a+1)
a = 7
b = 5
print(myFunction4(),a)
```

Exercise

- Without using IDLE predict the output of the fragment of code below (Local5.py)

```
def myFunction5():
    a += 1
    return(a)


a = 7
b = 5
print(myFunction5(),a)
```

Exercise

- Without using IDLE predict the output of the fragment of code below (Local6.py)

```
def myFunction6(x):
    lst1.append(x)
    return(b+1)


lst1=[1,2,3]
b = 5
print(myFunction6(5), lst1)
```

Exercise

- Without using IDLE predict the output of the fragment of code below (Local7.py)

```
def myFunction6(x):
    lst1=lst1.append(x)
    return(b+1)


lst1=[1,2,3]
b = 5
print(myFunction6(5), lst1)
```

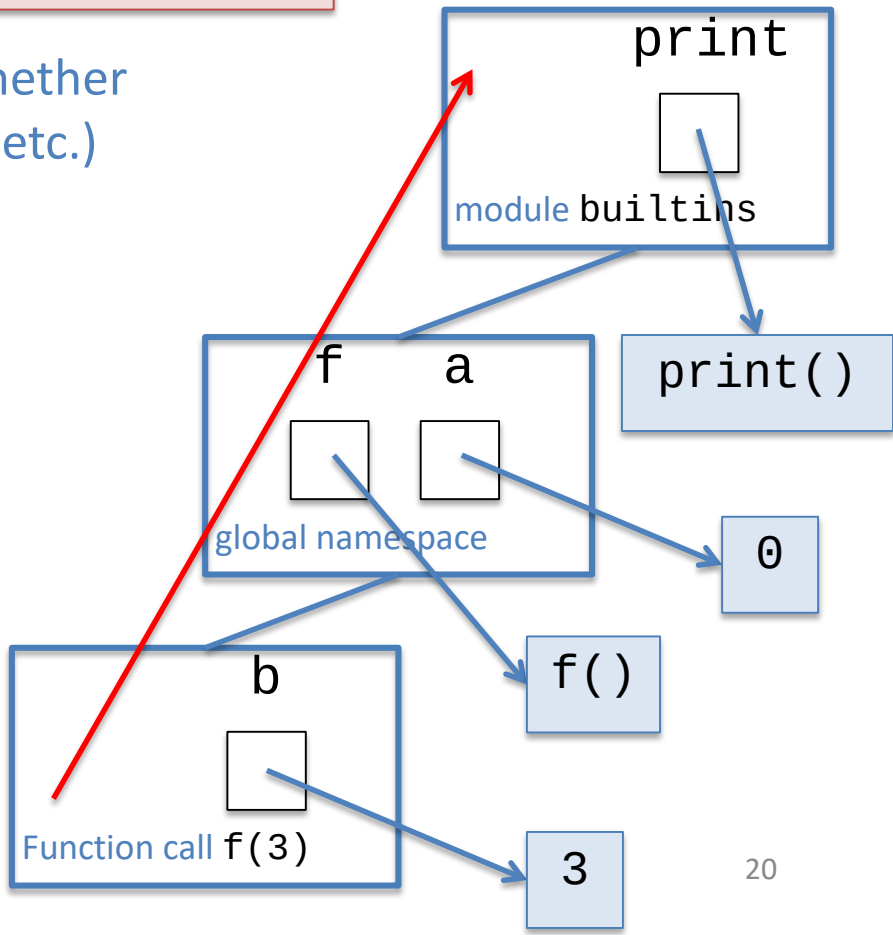
How Python evaluates names

```
def f(b):       # f has global scope, b has local scope
    return a*b  # this a is the global a

a = 0           # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a is still 0
```

How does the Python interpreter decide whether to evaluate a name (of a variable, function, etc.) as a local or as a global name?

Whenever the Python interpreter needs to evaluate a name, it searches for the name definition in this order:

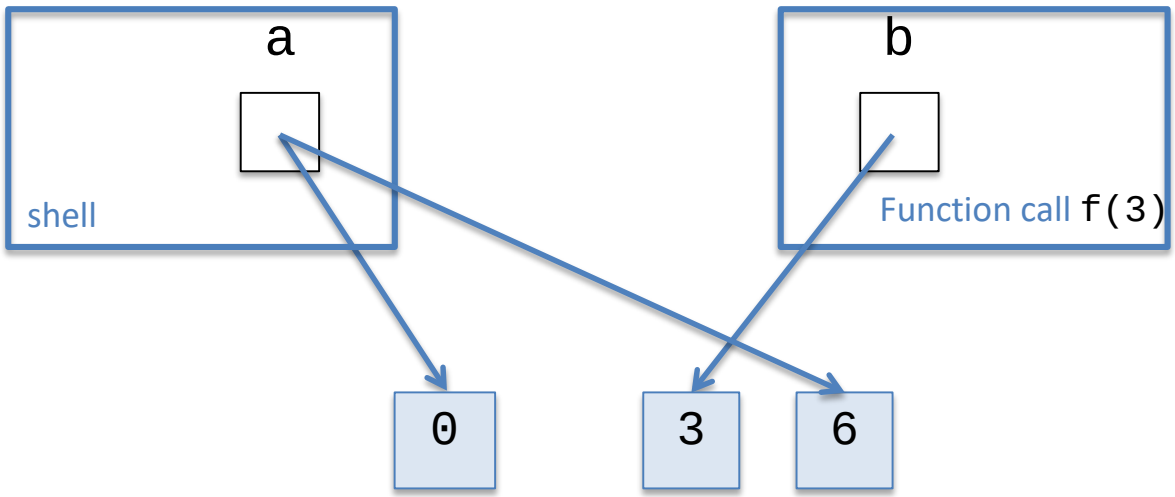
1. First the enclosing function call namespace
2. Then the global (module) namespace
3. Finally the namespace of module `builtins`



Modifying a global variable inside a function

```
def f(b):
    global a    # all references to a in f() are to the global a
    a = 6       # global a is changed
    return a*b  # this a is the global a

a = 0           # this a has global scope
print('f(3) = {}'.format(f(3)))
print('a is {}'.format(a))        # global a has been changed to 6
```

```
>>> === RESTART ====
>>>
f(3) = 18
a is 6
>>>
```



Encapsulation

- In a truly encapsulated code each function should be completely self standing, independent of the rest of the code
- Should communicate with the “outside” code by:
  - Accepting parameters with clear contracts between the pieces of code
  - Returning values with clear expectations

ERRORS AND EXCEPTIONS

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
```

Erroneous state errors

The program execution gets into an erroneous state

```
>>> 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'
```

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
(In some other languages the “throwing” of an exception)

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

Exercise

- Each of the statements to the right causes a syntax error or raises an exception
- Predict which statement will raise a syntax error and which will raise exceptions.
- For the latter predict the type of exception raised
- Execute the statements and verify your prediction

1. myList = [4,3,2,1]
   myList[4]
2. print('hello)
3. myString.count('love')
4. z + 3
5. a,b = 'one', 'two'
   a*b
6. float('one.two')
7. if x > 3
8. s = 'great'
   s[0] = 't'

Exceptions→ exceptional control flow

When the program execution gets into an erroneous state, an exception object is raised and 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

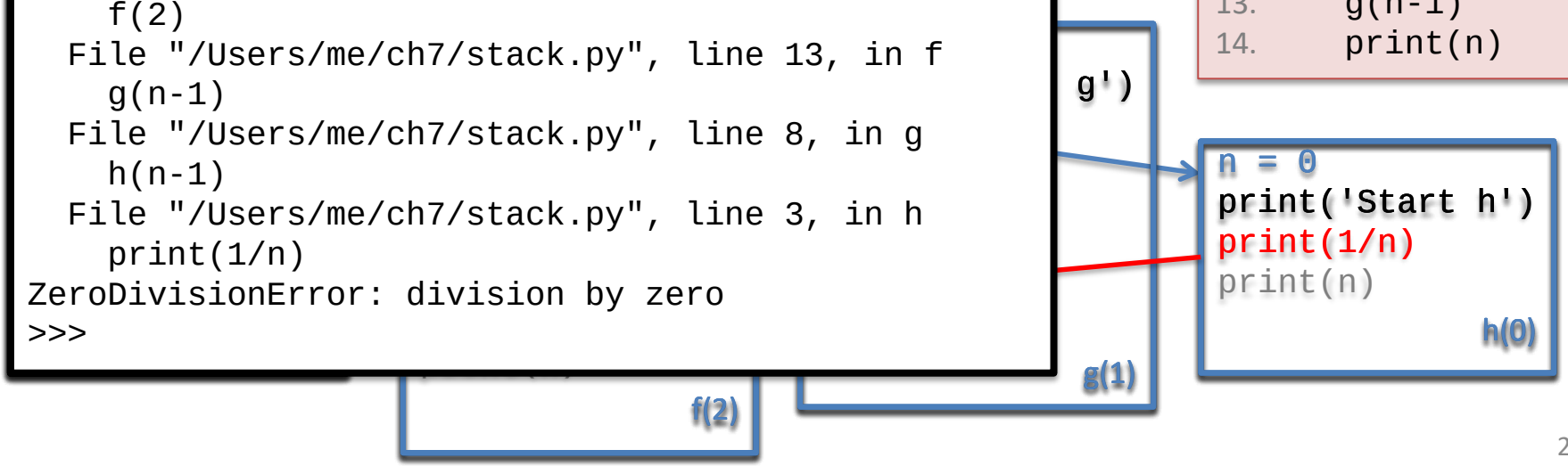
Exceptional control flow

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!')
```

Default behavior:

Custom behavior:

The `except` code block is the exception handler

```
>>> ========== RESTART ==========
>>>
Enter your age: fifteen
Enter your age using digits 0-9!
>>>
    intAge = int(strAge)
ValueError: invalid literal for int() with base 10: 'fifteen'
>>>
```

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

```
>>> 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'
>>>
```

default exception handler prints this

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.')
```

Exercise

- When we try to open a file often we are confronted by the raising of an IOError exception if the file is not found
- Write a function called safeOpen as a “wrapper” for the standard open function so that:
  - safeOpen takes two parameters, like open, a file name and a mode
  - If no exception is raised, the same file object generated by open is returned
  - If an IOError is raised then your function should return a None object reference
    - Simply use **`return None`**

Exercise

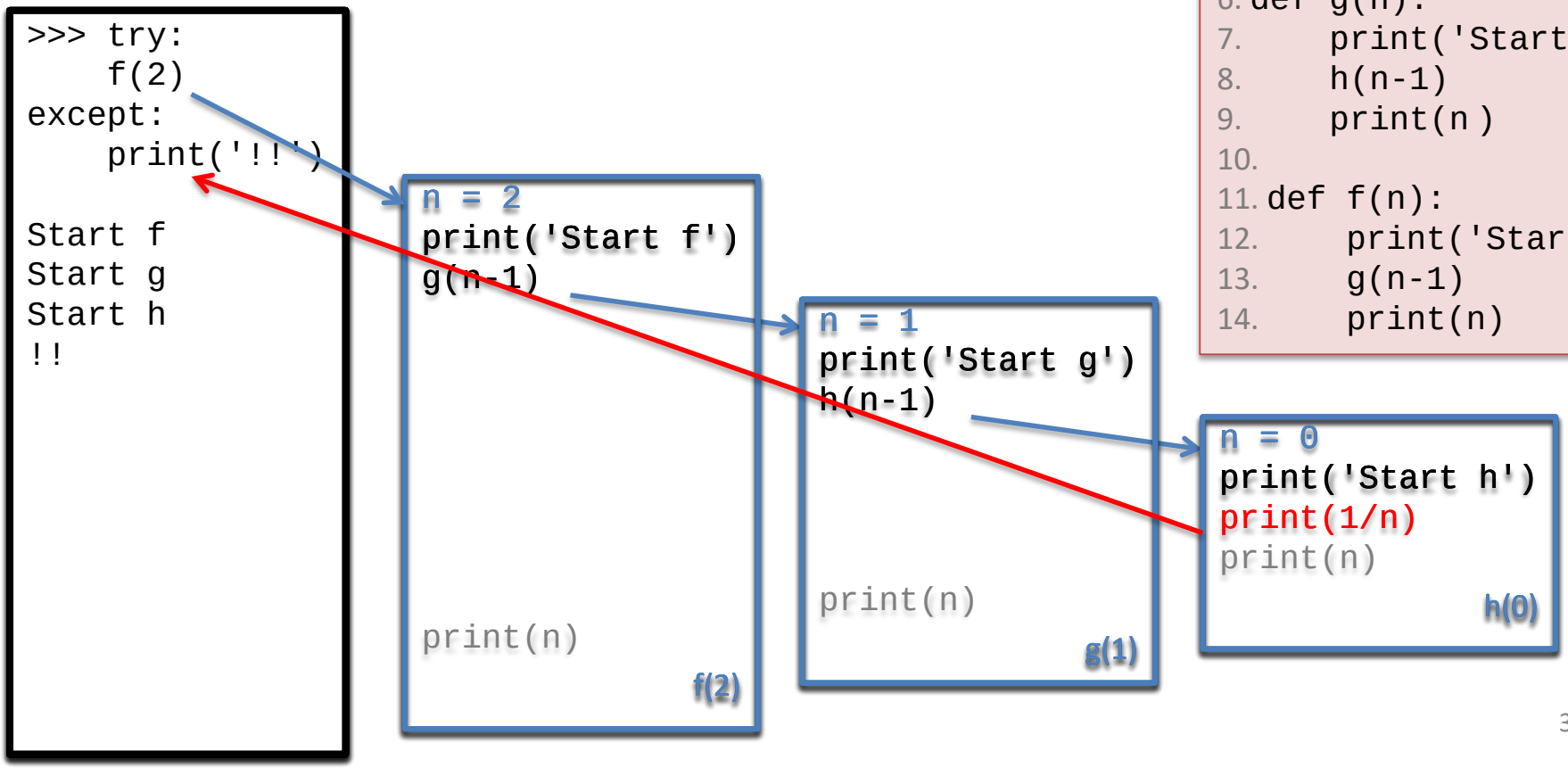
- Write a function called safeOpen2 that:
  - takes no parameters,
  - Asks the user for a file to open (in default reading mode)
  - If no exception is raised, the same file object generated by the regular open function is returned
  - If an IOError is raised then your function re-prompts the user for a file path

Exercise

- Write a function that asks a user to enter her age
- As long as the user enters something incorrect (e.g. the string ‘fifteen’) your function re-prompts
- Use try-except blocks

Controlling the exceptional control flow

```
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)
```



A different approach

- As the exception is raised in function h, and we understand fully what MAY go wrong, one could decide to catch the exception inside function h

```
def h(n):
  try:
     print('Start h')
     print(1/n)
     print(n)
  except:
     print(‘Something went wrong’)
```