Namespaces and Exceptions, revisited

- Encapsulation in Functions
- Global versus Local Namespaces
- Exceptional Control Flow
- Modules as Namespaces
- Classes as Namespaces

The purpose of functions

Wrapping code into functions has several desirable goals:

- Modularity: The complexity of developing a large program can be dealt with by breaking down the program into smaller, simpler, self-contained pieces. Each smaller piece (e.g., function) can be designed, implemented, tested, and debugged independently.

- Code reuse: A fragment of code that is used multiple times in a program—or by multiple programs— should be packaged in a function. The program ends up being shorter, with a single function call replacing a code fragment, and clearer, because the name of the function can be more descriptive of the action being performed by the code fragment. Debugging also becomes easier because a bug in the code fragment will need to be fixed only once.

- Encapsulation: A function hides its implementation details from the user of the function; removing the implementation details from the developer's radar makes her job easier.

Encapsulation through local variables

Encapsulation makes modularity and code reuse possible

```
>>> 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

```
>>> x, y = 20, 50
>>> res = double(5)
```

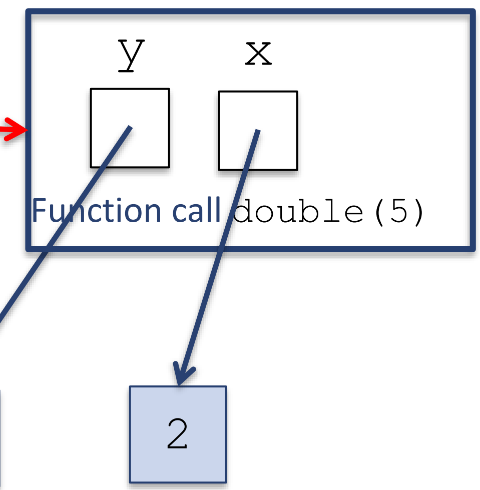
How is it possible that the values of `x` and `y` do not interfere with each other?

```
>>> 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
```

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

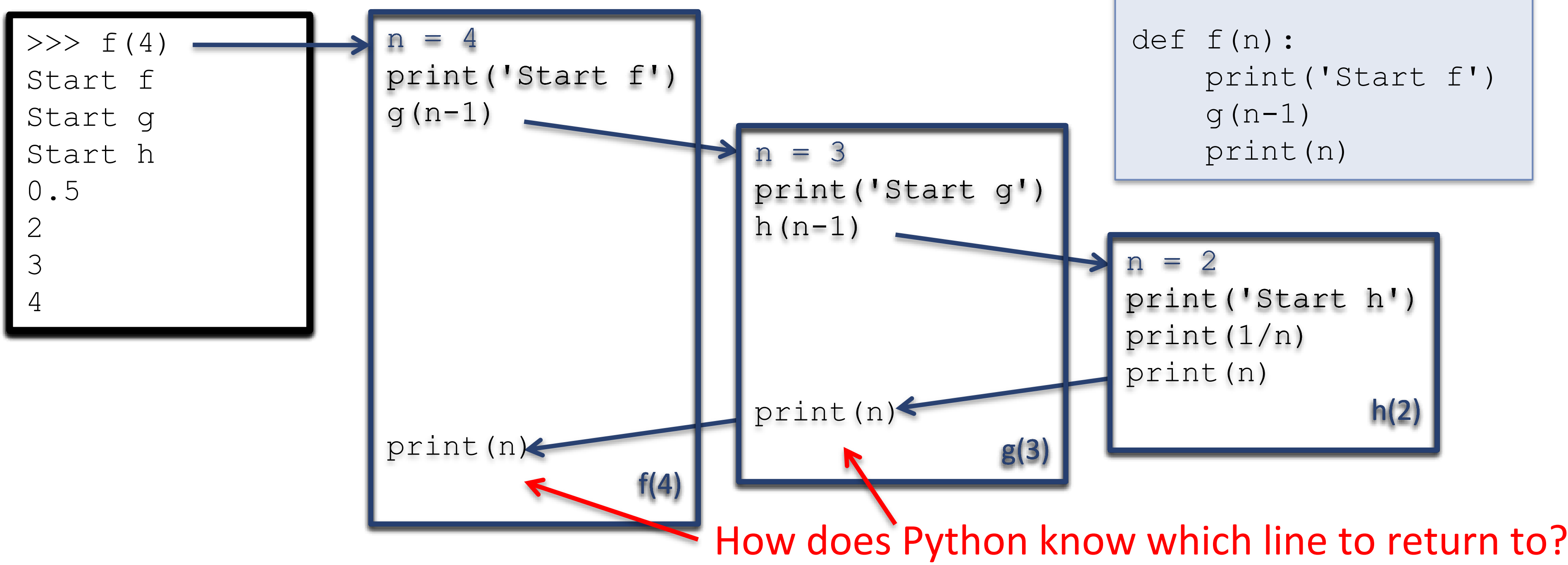
Every function call has a namespace in which local variables are stored

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)
```



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

```
line = 9
n = 3
line = 14
n = 4
```

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

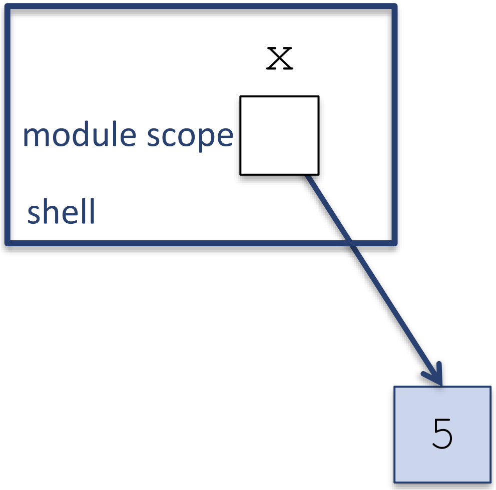
Every function call has a namespace associated with it.

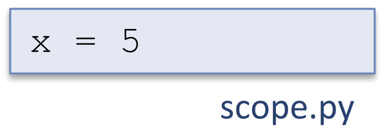
- This namespace is where names defined during the execution of the function (e.g., local variables) live.
- The scope of these names (i.e., the space where they live) is the namespace of the function.

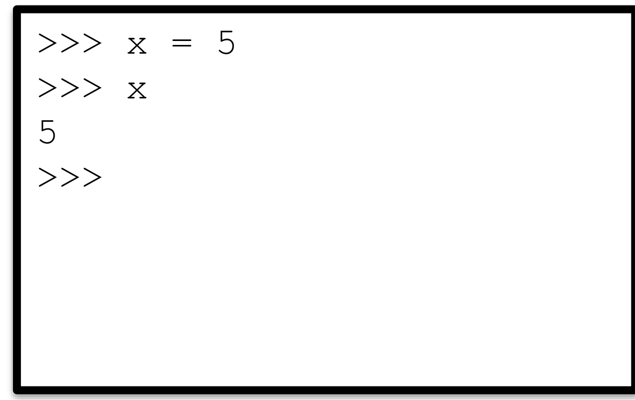
In fact, every name in a Python program has a scope

- 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







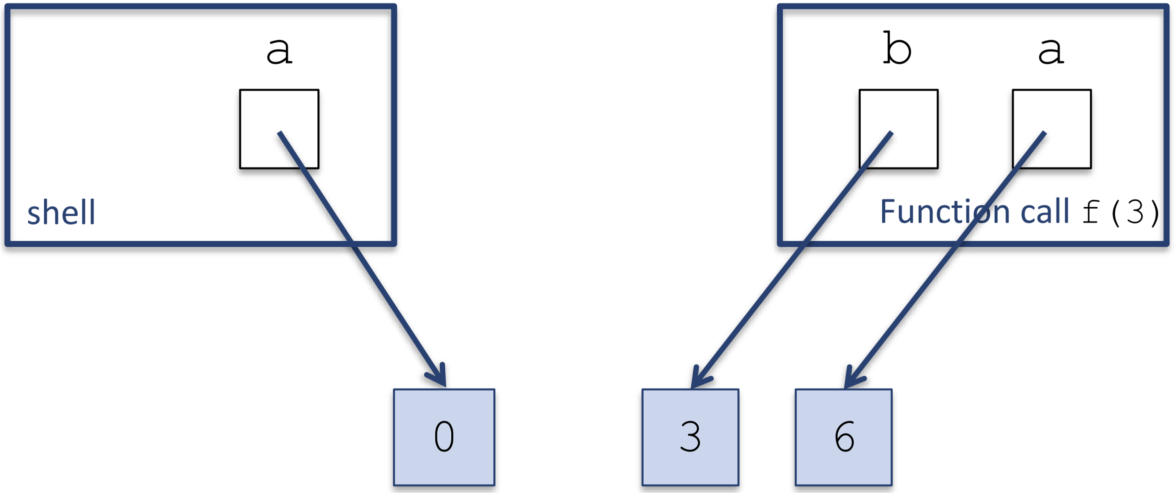
In fact, every name in a Python program has a scope

- 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
```

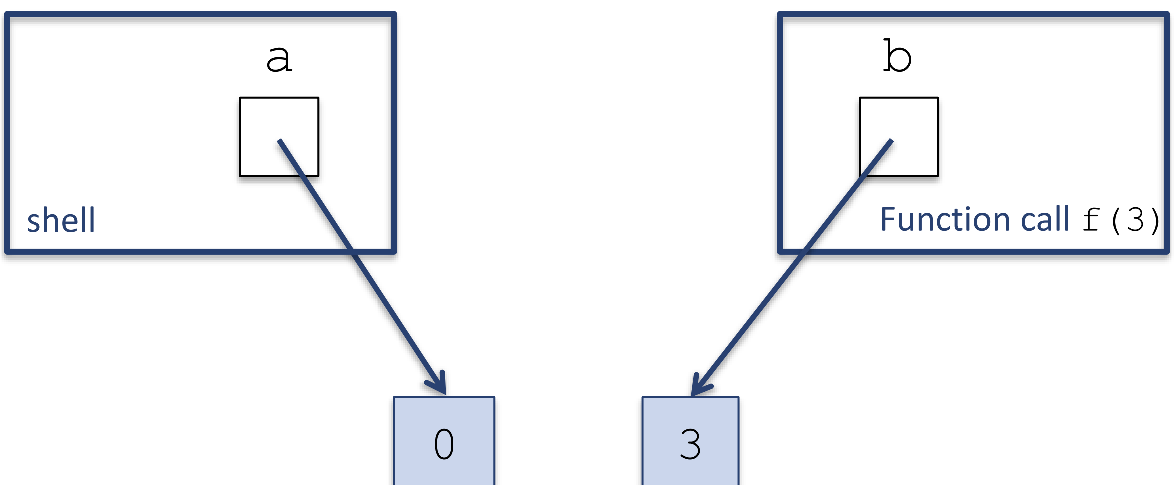



```
>>> === RESTART ====
>>>
f(3) = 18
a is 0
>>>
```

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
>>>
```

How Python evaluates names

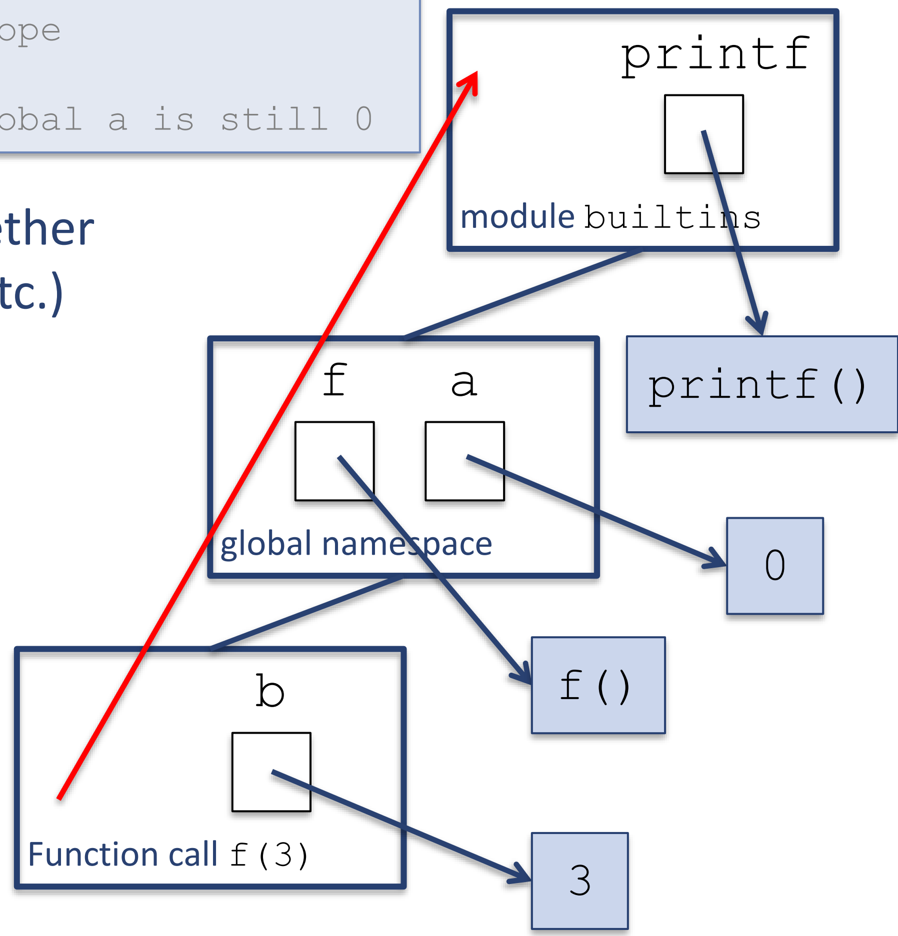
```
def f(b):          # f has global scope, b has local scope
    return a*b     # this a is the global a

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:

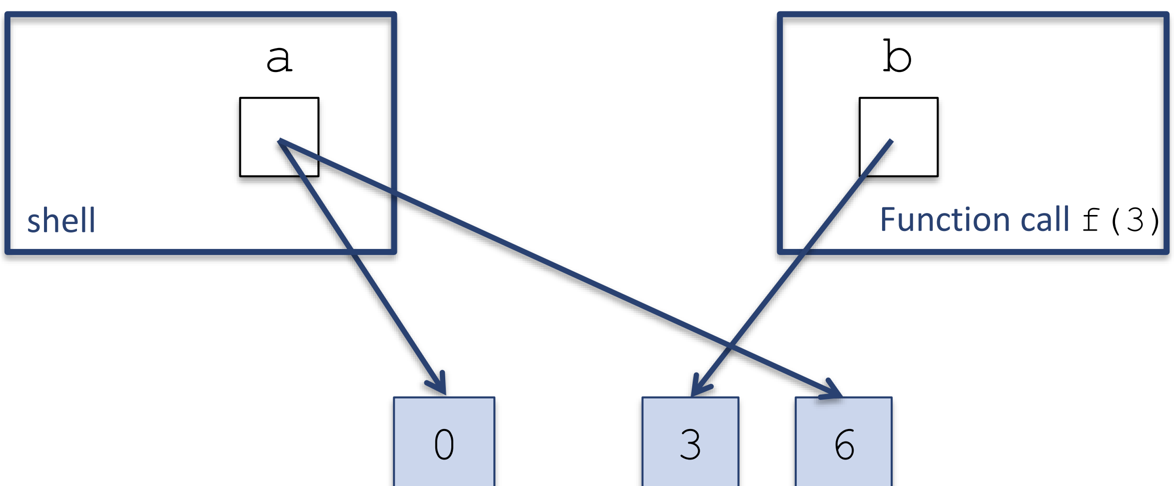
1. First the enclosing function call namespace
2. Then the global (module) namespace
3. Finally the namespace of module `builtins`



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
>>>
```

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

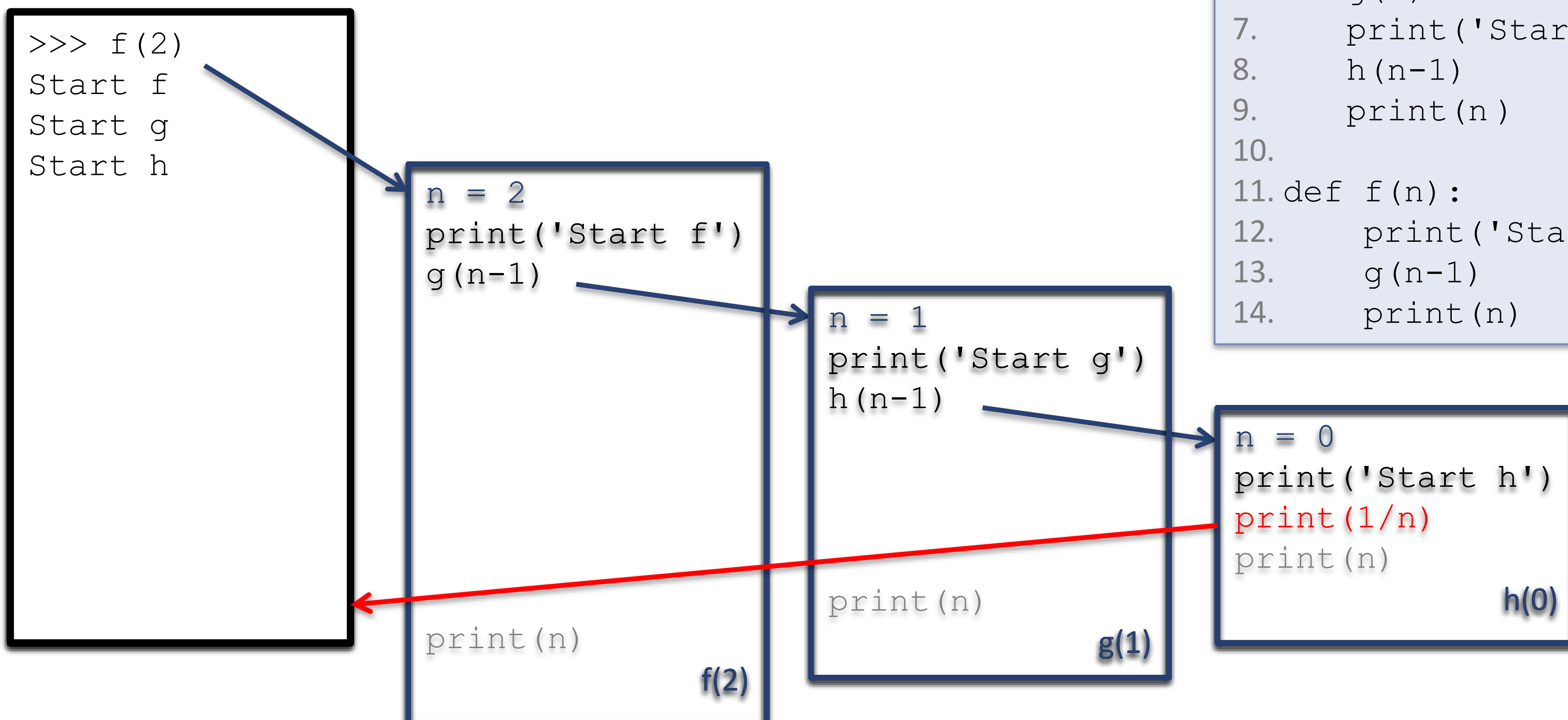
Normal control flow

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

```
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)
```



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

```
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!')
```

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

The `except` code block is the exception handler

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

Default behavior:

```
>>> ================
>>>
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'
>>>
```

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

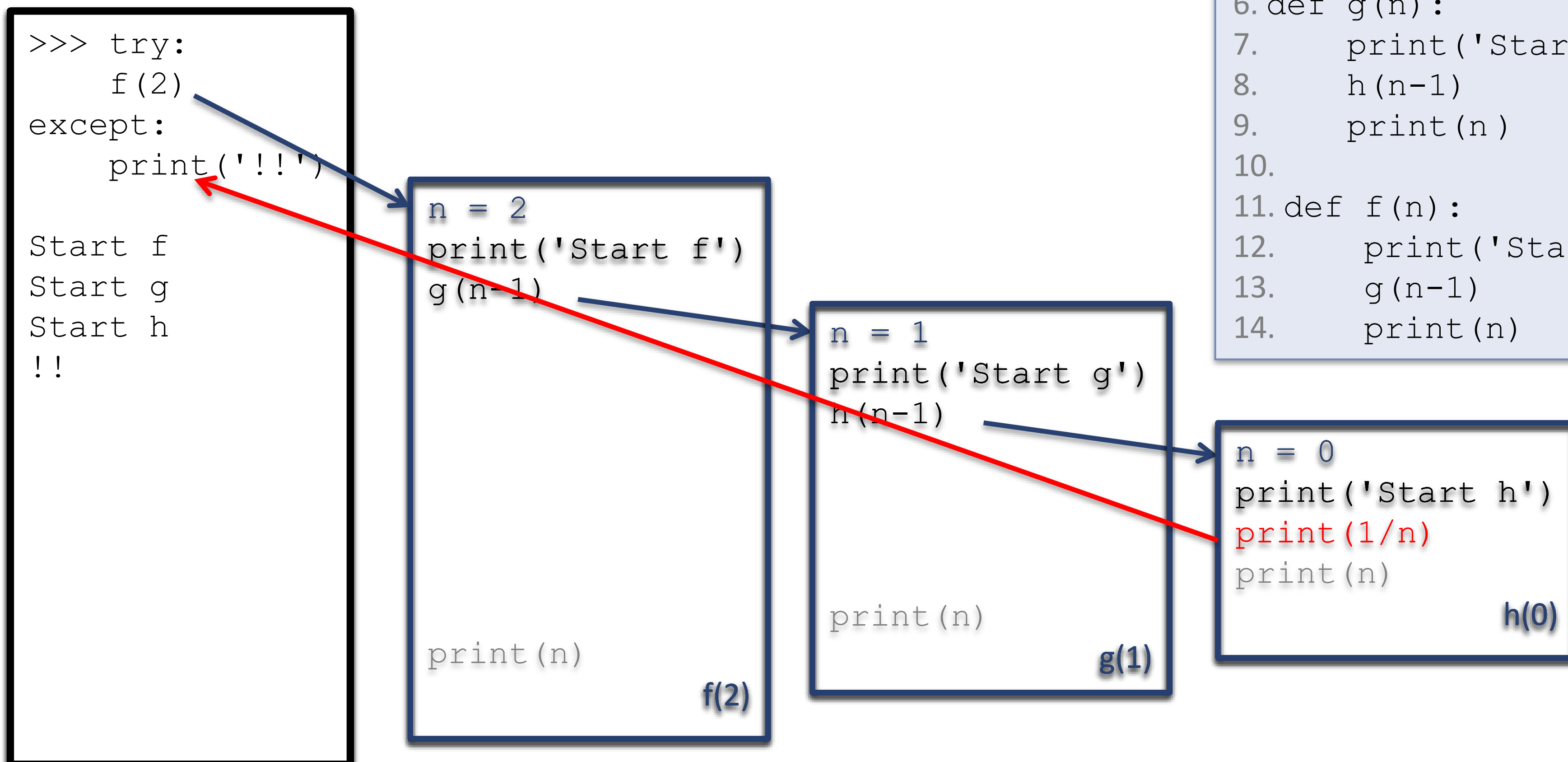
```
def readAge(filename):
    'converts first line of file filename to an integer and prints it'
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is',age)
    except IOError:
        # executed only if an IOError exception is raised
        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.')
```

Controlling the exceptional control flow

```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)
```



Modules, revisited

A module is a file containing Python code.

When the module is executed (imported), then the module is (also) a namespace.

- This namespace has a name, typically the name of the module.
- In this namespace live the names that are defined in the global scope of the module: **the names of functions, values, and classes defined in the module.**
- These names are the module's attributes.

Built-in function `dir()` returns the names defined in a namespace

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> math.sqrt
<built-in function sqrt>
>>> math.pi
3.141592653589793
```

To access the imported module's attributes, the name of the namespace must be specified

Importing a module

When the Python interpreter executes an `import` statement, it:

1. Looks for the file corresponding to the module to be imported.
2. Runs the module's code to create the objects defined in the module.
3. Creates a namespace where the names of these objects will live.

An import statement only lists a name, the name of the module

- without any directory information or .py suffix.

Python uses the Python search path to locate the module.

- The search path is a list of directories where Python looks for modules.
- The variable name `path` defined in the Standard Library module `sys` refers to this list.

current working directory   Standard Library folders

```
>>> import sys
>>> sys.path
['/Users/me', '/Library/Frameworks/Python.framework/Versions/3.2/lib/python32.zip',
. . .
'/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages']
>>>
```

The Python search path

Suppose we want to import module `example` stored in folder `/Users/me` that is not in list `sys.path`

By just adding folder `/Users/me` to the search path, module `example` can be imported

no folder in the Python search path contains module `example`

names in the shell namespace; note that `example` is not in

```
>>> ================ RESTART ================
>>> dir()
['__builtins__', '__doc__', '__name__',
'__package__']
>>> import example
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    import example
ImportError: No module named example
>>> import sys
>>> sys.path.append('/Users/me')
>>> import example
>>> example.f
<function f at 0x10278dc88>
>>> example.x
0
>>> dir()
['__builtins__', '__doc__', '__name__',
'__package__', 'example', 'sys']
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

When called without an argument, function `dir()` returns the names in the top-level module

- the shell, in this case.

Top-level module

A computer application is a program typically split across multiple modules.

One of the modules is special: It contains the "main program". This module is referred to as the top-level module.

- The remaining modules are "library" modules that are imported by other modules and that contain functions and classes used by it

When a module is imported, Python creates a few "bookkeeping" variables in the module namespace, including variable `__name__`:

- set to `'__main__'`, if the module is being run as a top-level module

A module is a top-level module if:

- it is run from the shell
- it is run at the command line

```
print('My name is {}'.format(__name__))
```

name.py

```
> python name.py
My name is __main__
```

Top-level module

A computer application is a program typically split across multiple modules.

One of the modules is special: It contains the "main program". This module is referred to as the top-level module.

- The remaining modules are "library" modules that are imported by the top-level module and that contain functions and classes used by it

When a module is imported, Python creates a few "bookkeeping" variables in the module namespace, including variable `__name__`:

- set to `'__main__'`, if the module is being run as a top-level module
- set to the module's name, if the file is being imported by another module

```
print('My name is {}'.format(__name__))
```

name.txt

```
>>> === RESTART ===
>>>
My name is name
```

```
import name
```

import.txt

Three ways to import module attributes

1. Import the (name of the) module

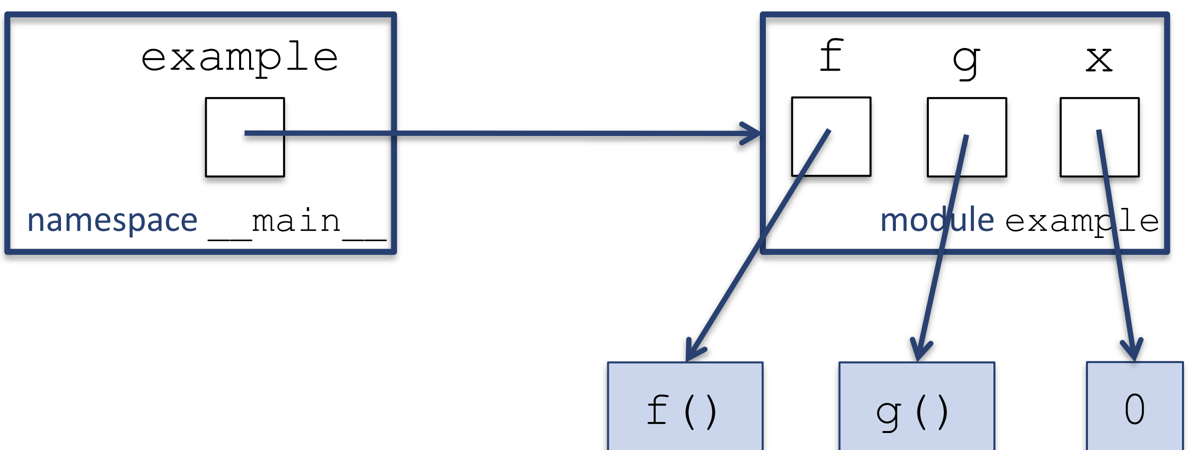
```
>>> import example
>>> example.x
0
>>> example.f
<function f at 0x10278dd98>
>>> example.f()
Executing f()
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



Three ways to import module attributes

2. Import specific module attributes

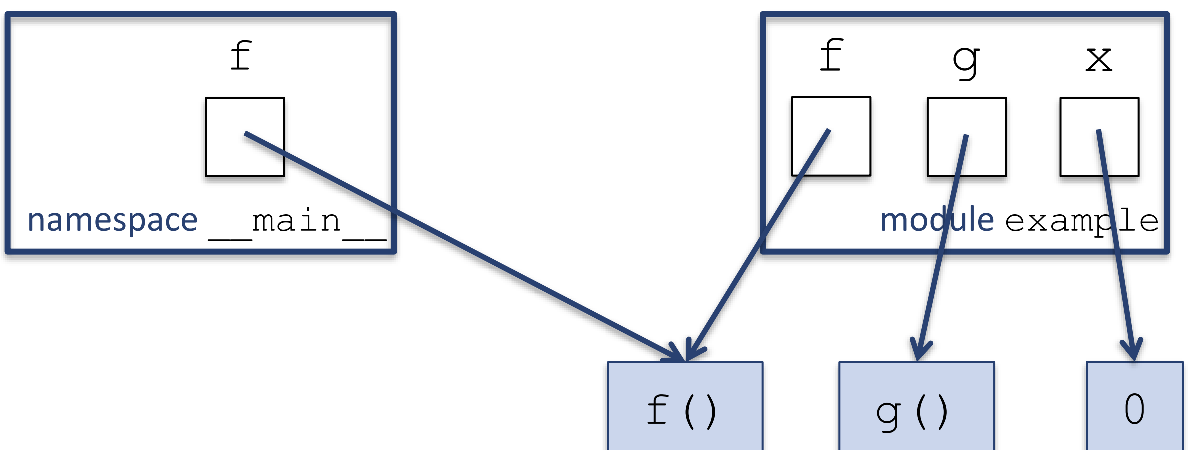
```
>>> from example import f
>>> f()
Executing f()
>>> x
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



Three ways to import module attributes

3. Import all module attributes

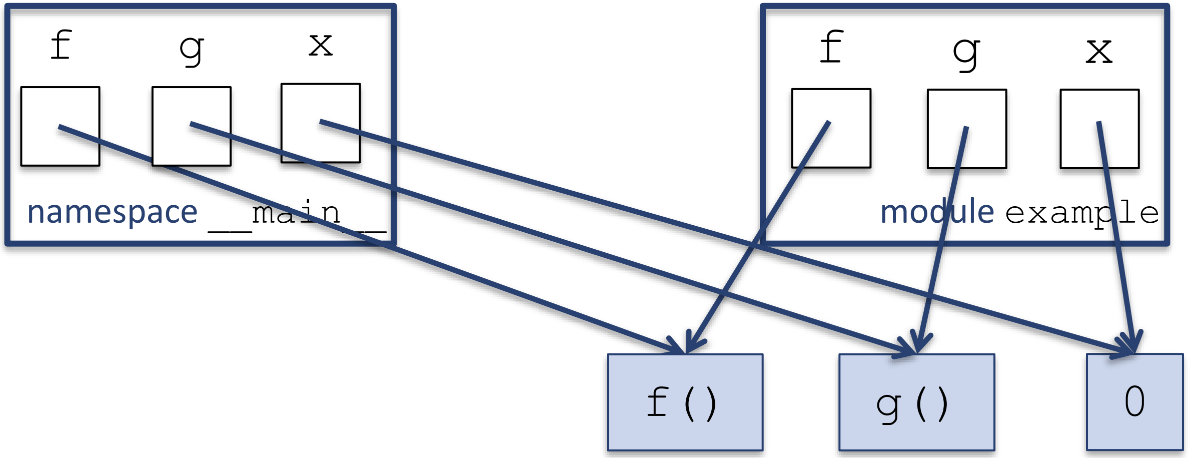
```
>>> from example import *
>>> f()
Executing f()
>>> g()
Executing g()
>>> x
0
>>>
```

```
'an example module'
def f():
    'function f'
    print('Executing f()')

def g():
    'function g'
    print('Executing g()')

x = 0  # global var
```

example.txt



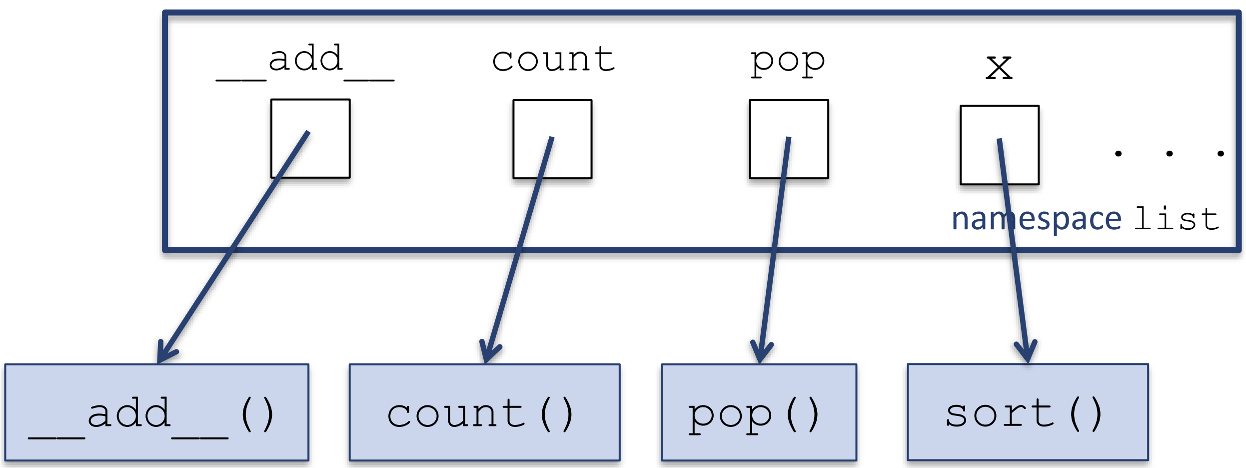
A class is a namespace

A class is really a namespace

- The name of this namespace is the name of the class
- The names defined in this namespace are the class attributes (e.g., class methods)
- The class attributes can be accessed using the standard namespace notation

```
>>> list.pop
<method 'pop' of 'list' objects>
>>> list.sort
<method 'sort' of 'list' objects>
>>> dir(list)
['__add__', '__class__',
...
'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Function `dir()` can be used to list the class attributes



Class methods

A class method is really a function defined in the class namespace; when Python executes

```
instance.method(arg1, arg2, …)
```

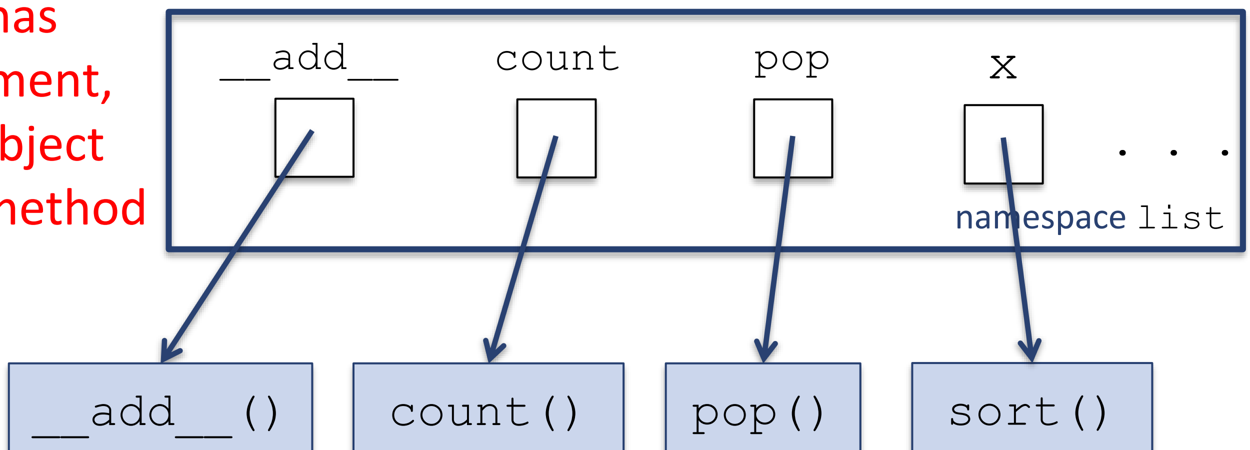
it first translates it to

```
class.method(instance, arg1, arg2, …)
```

and actually executes this last statement

```
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> lst.sort()
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> list.sort(lst)
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst.append(6)
>>> lst
[1, 2, 3, 7, 8, 9, 6]
>>> list.append(lst, 5)
>>> lst
[1, 2, 3, 7, 8, 9, 6, 5]
```

The function has an extra argument, which is the object invoking the method



Exercise

Rewrite the below Python statement so that instead of making the usual method invocations

```
instance.method(arg1, arg2, …)
```

you use the notation

```
class.method(instance, arg1, arg2, …)
```

```
>>> s = 'hello'
>>> s = 'ACM'
>>> s.lower()
'acm'
>>> s.find('C')
1
>>> s.replace('AC', 'IB')
'IBM'
```

```
>>> s = 'ACM'
>>> str.lower(s)
'acm'
>>> str.find(s, 'C')
1
>>> str.replace(s, 'AC', 'IB')
'IBM'
>>>
```