CS100 Python Introduction to Programming

Lecture 23. Exception, Function and Scope

Fu Song

School of Information Science and Technology

ShanghaiTech University

Learning Objectives

- **Exception**
- **Function**
  - Function definition
  - Function invocation
- **Scope**
  - Local
  - Non-local
  - Global
  - Name resolution

The assert statement

```
def Div(x,y):
    assert y!=0, "denominator is 0"
    return x/y

x = int(input("Input numerator:"))
y = int(input("Input denominator:"))
print(Div(x,y))
```

Input/Output

```
Input numerator:1
Input denominator:0
Traceback (most recent call last):
...
  File "C:\Users\Fu Song\Desktop\hello.py", line 2, in Div
    assert y!=0, "denominator is 0"
AssertionError: denominator is 0
```

Errors and Exceptions

There are (at least) two distinguishable kinds of errors

- Syntax errors: a.k.a. parsing errors, most common one

```
>>> while True print('Hello world')
File "<stdin>", line 1
while True print('Hello world')
^
SyntaxError: invalid syntax
```

- Exceptions: errors detected during execution

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Handling Exceptions

- Improve robustness and fault tolerance
- User-friendly error message
- Try statement

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number.
Try again...")
```

If-Else vs. Exception Handling

- It is better to use exception handling than if-else
- Proper use of exception handling, instead of abuse of exception handling
- Catch precise exception
- Proper exception handling for different exception

Exception hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

The try statement

```
try_stmt  ::=  try1_stmt | try2_stmt

try1_stmt ::=  "try" ":" suite
               "finally" ":" suite

try2_stmt ::=  "try" ":" suite
               "except" [expression ["as" identifier]] ":" suite
                                .....
               "except" [expression ["as" identifier]] ":" suite
               ["else" ":" suite]
               ["finally" ":" suite]
```

The try statement

```
try:
    suite
finally:
    suite
```

- There is no exception handler
- But, the finally suite is always executed before leaving the try statement
- The finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except [expression [as en]]:  suite
[else:   suite]
[finally:   suite]
```

- If there no finally clause, then it contains at least one except clause
- The last except can omit the expression, in this case, it will catch all the possible exception

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except expression [as en]:  suite
[else:   suite]
[finally:   suite]
```

- If no exception occurs in the try clause, no exception handler is executed,
- **but** else clause is executed if no return, continue, or break statement was executed in try clause
- Exceptions in the else clause are not handled by the preceding except clauses

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except expression [as en]:  suite
[else:   suite]
[finally:   suite]
```

When an exception occurs in the try suite, a search for an exception handler is started from the except clauses in turn until one is found that matches the exception (type-checking)

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except expression [as en]:  suite
[else:   suite]
[finally:   suite]
```

If no except clause matches the exception in the current try block, the search continues in the surrounding code and on the invocation stack

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except expression [as en]:  suite
[else:   suite]
[finally:   suite]
```

- When a matching except clause is found, the except clause's suite is executed
- When the end of this block is reached, execution continues normally after the entire try statement

The try statement

```
try:  suite
except expression [as e1]:  suite
   ……
except expression [as en]:  suite
[else:   suite]
[finally:   suite]
```

- If finally is present, it is always executed
- Moreover, during the search in the surrounding code and on the invocation stack, finally clause of the searching try block is also executed whatever exception is matched or not

The try statement

```
def foo(a,b):
    try:
        print("try-1")
        x = a/b
        print("try-2")
    except ZeroDivisionError:
        print("except")
        return
    else:
        print("else")
        return
    finally:
        print("finally")

foo(1,2)
```

Output

```
try-1
try-2
else
finally
>>>
```

If finally is present, it is always executed

The try statement

```
def foo(a,b):
    try:
        print("try-1")
        x = a/b
        print("try-2")
    except ZeroDivisionError:
        print("except")
        return
    else:
        print("else")
        return
    finally:
        print("finally")

foo(1,0)
print("after foo")
```

Raise here

Output

```
try-1
except
finally
after foo
>>>
```

If finally is present, it is always executed even if there is return

The try statement

```
def foo(a,b):
    try:
        print("try-1")
        x = a/b
        print("try-2")
    except AssertionError:
        print("except")
        return
    else:
        print("else")
        return
    finally:
        print("finally")

foo(1,0)
```

Output

```
try-1
finally
Traceback (most recent
call last):
  ...
ZeroDivisionError:
division by zero
>>>
```

The try statement

```
def foo(a,b):
    try:
        print("try-1")
        x = a/b
        print("try-2")
    except AssertionError:
        print("except-1")
        return
    finally:
        print("finally-1")
try:
    foo(1,0)
except ZeroDivisionError:
    print("except-2")
else:
    print("else-2")
finally:
    print("finally-2")
```

Output

```
try-1
finally-1
except-2
finally-2
>>>
```

Surrounding finally is also executed

The try statement

```
def foo(a,b):
    try:
        print("try-1")
        x = a/b
        print("try-2")
    except AssertionError:
        print("except-1")
        return
    finally:
        print("finally-1")
try:
    foo(1,0)
except IndexError:
    print("except-2")
else:
    print("else-2")
finally:
    print("finally-2")
```

Output

try-1
finally-1
finally-2
Traceback (most recent call last):
…
ZeroDivisionError: division by zero

Surrounding finally is also executed

Exception hierarchy

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

The try statement

```
x = [1,2,3]
try:
print(x[0])
print(x[3])
except IndexError:
print("Out of range")

try:
print(x[0])
print(x[3])
except LookupError:
print("Out of range")
```

Output

```
1
Out of range
1
Out of range
>>>
```

The more specific exception handler, the better

Learning Objectives

- **Exception**
- **Function**
  - Function definition
  - Function invocation
- **Scope**
  - Local
  - Non-local
  - Global
  - Name resolution

Function definition

funcdef ::=
“def" funcname "(" [parameter_list] ") " ":" suite

- No return type or types of parameters
- Don’t miss colon (:)
- Don’t miss INDENT in the function body, i.e., suite
- Nested function definitions are allowed
- Function definition itself is an object

Function definition: Example

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
fib(1000)
```

Compute fibonacci sequence

Function definition: docstring

funcdef ::=
“def" funcname "(" [parameter_list] ") " ":" suite

- We can add a comment after : in function definition in the form of

  ‘’’multiple comments’’’

- Docstring will regard the comment as the information of the function
- It is better to add function comment in your code

Function definition: Example

```
#fib.py
def fib(n):
    '''Input: an integer n>=0
       Output: Fibonacci sequence'''
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
fib(1000)
```

```
>>> import os
>>> os.chdir("D:\\Test")
>>> import fib
0 1 1 2 3 5 8 13 21 34 55 89 144
Help on function fib in module fi
```

import os package and set the path of console to path of fib.py

```
fib(n)
    Input: an integer n such that n>=0
    Output: the numbers less than n in Fibonacci sequence
```

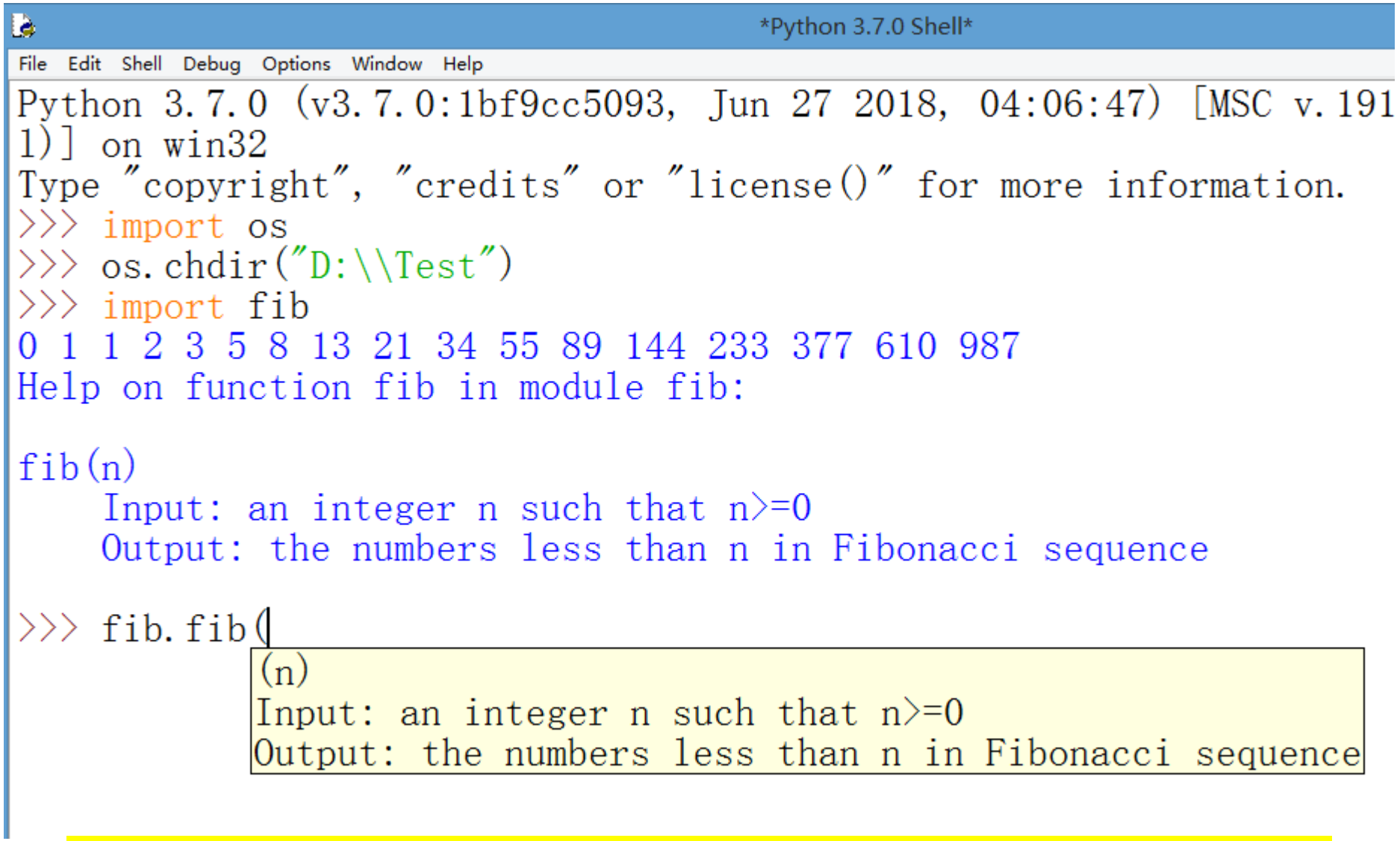
```
>>> help(fib)
Help on module fib:

NAME
    fib

FUNCTIONS
    fib(n)
        Input: an integer n such that n>=0
        Output: the numbers less than n in Fibonacci sequence

FILE
    d:\test\fib.py
```

Help(fib) also shows information of fib.py



Type left (shows information of the function

Function definition is object

```
>>> import os
>>> os.chdir("D:\\Test")
>>> import fib
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
Help on function fib in module fib:

fib(n)
    Input: an integer n such that n>=0
    Output: the numbers less than n in Fibonacci sequence

>>> type(fib.fib)
<class 'function'>
>>>
```

The type of a function object is function

Can we add types of the return value and parameters in a function definition?

Type annotation

Types of parameters type

```
def GetLarger(x:int, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
```

return type

Does it work?

Type annotation

```
def GetLarger(x:int, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
print(GetLarger(1,2))
print(GetLarger(1,2.0))
```

Output

```
2
2.0
```

Type annotation does not make any sense

It is still recommended to add for readability

Default values of arguments

- Parameters can have default values,

p1 =v1, p2 = v2,…, pn = vn

- But, all the parameters occurred at right hand of some parameter having default value must have default values

Default values of arguments

```
def GetLarger(x = 1, y = 2):
    if x>=y:
        return x
    else:
        return y
```

```
def GetLarger(x:int, y:int=2) -> int:
    if x>=y:
        return x
    else:
        return y
```

No error

Default values of arguments

```
def GetLarger(x = 1, y):
    if x>=y:
        return x
    else:
        return y
```

```
def GetLarger(x:int=1, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
```

SyntaxError: non-default argument follows default argument

Function call

```
def  funcname(p1,p2,…,pn):
    body
```

Actual arguments in function call can be passed by

- **positional argument**: argument are passed in the order of parameters

  funcname(v1,v2,…,vn)

- **keyword argument**: argument are passed by preceding the corresponding parameters without preserving their orders

  funcname(pi=vj,pj=vj,…)

Function call: Example

```
def GetPow(base, exp):
    '''input: two integers base and exp
       return  base**exp'''
    return base**exp

print(GetPow(2,3))
print(GetPow(exp = 3,base = 2))
```

Output

```
8
8
```

Arbitrary number of parameters

Arbitrary Argument List: a way to define a function with arbitrary number of parameters:

funcname(v_1,…,v_n, *para, x_1,…,x_m)

- zero or more normal arguments may occur for *para
- para is used as a tuple (tuple will be introduced later)
- Parameters x_1,…,x_m after *para should be used as keyword arguments

Arbitrary Argument List: Example

tuple

```
>>> def demo(*p):
    print(p)

>>> demo(1,2,3)
(1, 2, 3)
>>> demo(1,2)
(1, 2)
>>> demo(1,2,3,4,5,6,7)
(1, 2, 3, 4, 5, 6, 7)
>>> demo()
()
```

Tuple is in the form of (…)

Arbitrary Argument List: Example

```
>>> def demo(*p,v):
    print(p,v)

>>> demo(1,2,3,v=4)
(1, 2, 3) 4
>>> demo(1,2,3)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    demo(1,2,3)
TypeError: demo() missing 1 required keyword-
only argument: 'v'
>>>
```

Type error

Arbitrary number of parameters

Unpacking Argument List : another way to define a function with arbitrary number of parameters:

funcname(v_1,…,v_n, **para)

- zero or more keyword arguments may occur for **para
- para is used as a dictionary dict (introduced later)
- **para should be the last parameter
- v_n could be arbitrary argument list

Arbitrary Argument List: Example

```
>>> def demo(**p):
    print(p)

>>> demo(x=1,y=2)
{'x': 1, 'y': 2}
>>> demo(x=1,y=2,z=3)
{'x': 1, 'y': 2, 'z': 3}
>>>
```

keyword arguments

Dict

Arbitrary Argument List: Example

```
>>> def demo(**p,v):
    print(p)

SyntaxError: invalid syntax
>>>
```

SyntaxError

```
>>> def demo(**p,v=1):
    print(p)

SyntaxError: invalid syntax
>>>
```

Arbitrary Argument List: Example

```
>>> def demo(x,y,*t,**d):
    print(x,y,t,d)

>>> demo(1,2,3,4,a=5,b=6)
1 2 (3, 4) {'a': 5, 'b': 6}
>>>
```

Mixed use of Arbitrary Argument List and Unpacking Argument List

Lambda expression

lambdaexpr ::= “lambda" parameter_list" ":" expr

- Lambda expressions can be used to define small anonymous functions
- Parameter_list: is a list of parameters $p_1, p_2, \ldots, p_n$
- The anonymous function return the value of expr
- Parameters are similar to normal functions
  - Default value, Keyword arguments
  - Arbitrary Argument List and Unpacking Argument List

Lambda expression

```
>>> SumAll = lambda x,y,z:x+y+z
>>> type(SumAll)
<class 'function'>
>>> SumAll(1,2,3)
6
>>>
```

Lambda expression

```
>>> f = lambda x, y, z: x+y+z
>>> f(1,2,3)
6
>>> g = lambda x, y=2, z=3: x+y+z
>>> g(1)
6
>>> g(2, z=4, y=5)
11
```

Parameters are similar to normal functions

Nested Functions

Functions can be defined in a function body

```
def maker(n):
    def action(x):
        return x ** n
    return action
f = maker(2)
print(f)
print(f(3))
```

Function action is defined in the function maker

Output

```
<function maker.<locals>.action at 0x00C6B8E8>
9
>>>
```

Learning Objectives

- **Exception**
- **Function**
  - Function definition
  - Function invocation
- **Scope**
  - Local
  - Non-local
  - Global
  - Name resolution

Scope

- A scope defines the visibility of a name (variable, function, etc.) within a block
- If a local name is defined in a block, its scope includes that block
- If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name
- When a name is used in a code block, it is resolved using the nearest enclosing scope
- When a name is not found at all, a NameError exception is raised
- Scopes are either non-overlapped or nested

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

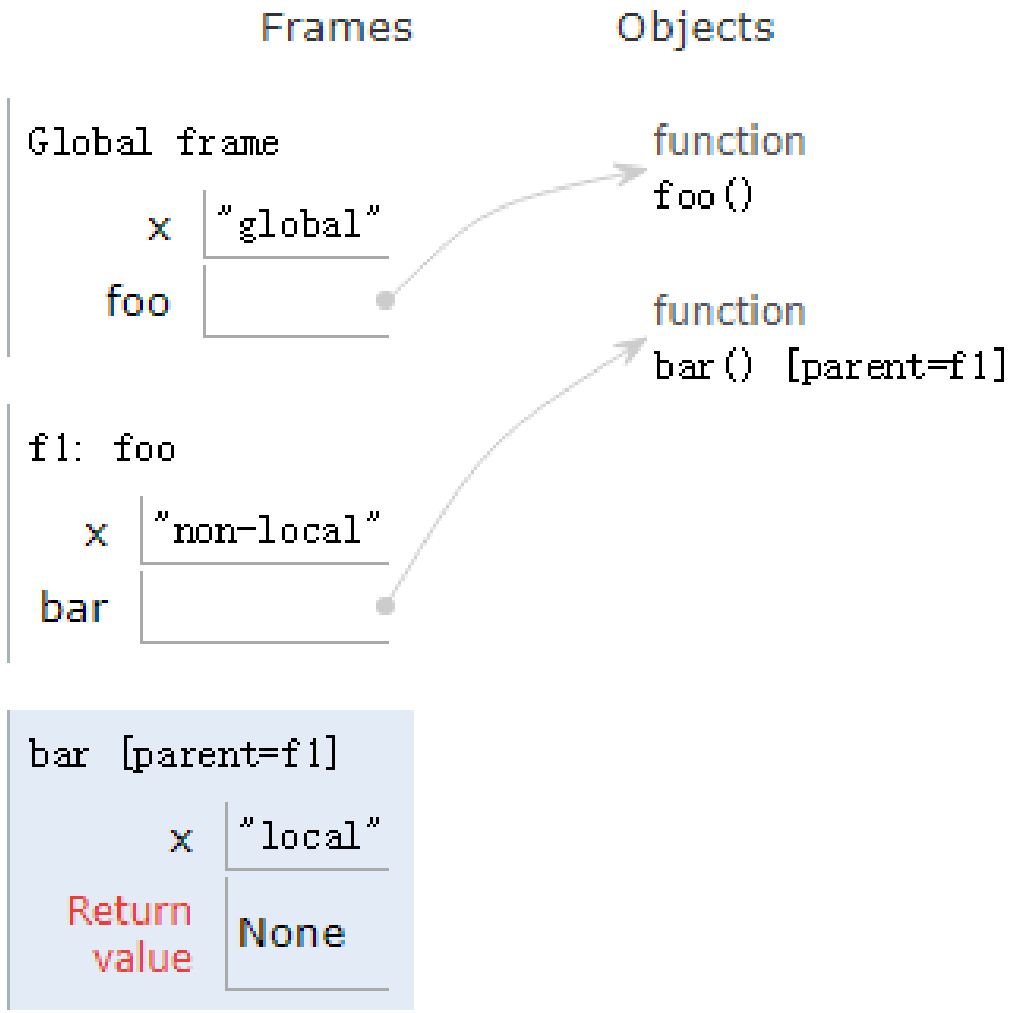
- global x: global - foo
- non-local x: foo - bar
- local x: bar

Output

```
global
non-local
local
>>>
```

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```



When a **name** is used in a code block, it first search the name in the current scope

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

- global x: global – foo
- non-local x: foo

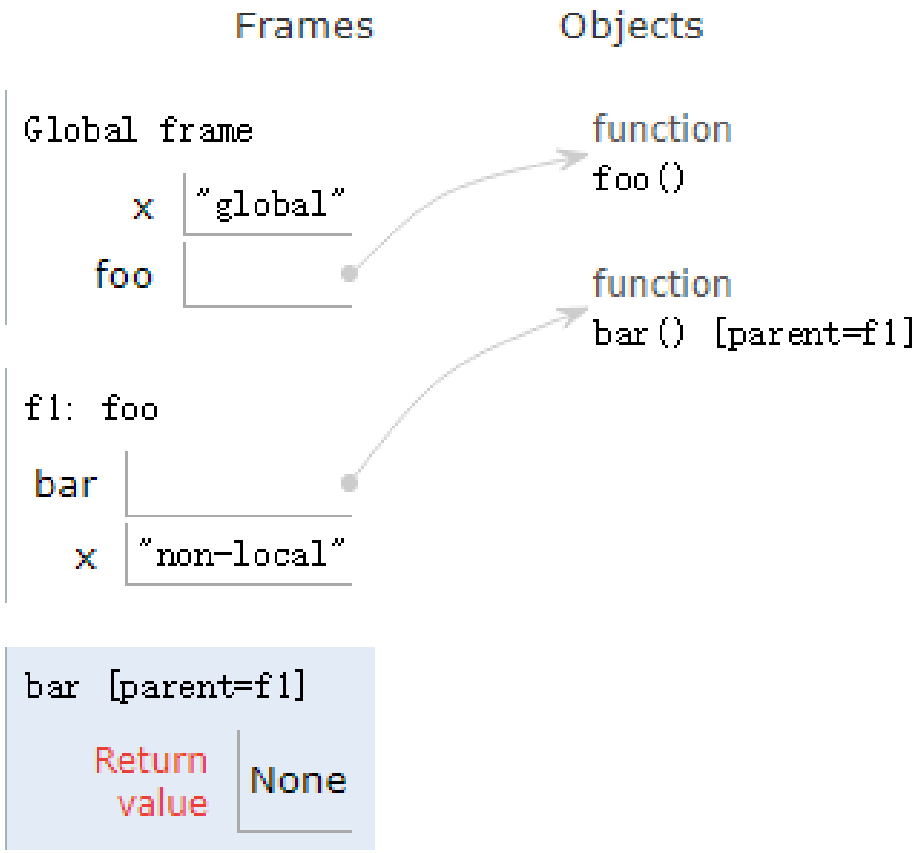
Output

```
global
non-local
non-local
>>>
```

- When a **name** is not found in the current block, it is resolved using the **nearest enclosing scope**

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
    print(x)
    bar()
print(x)
foo()
```



- print(x) in bar uses non-local x, as the scope of bar does not have x

Scope

```
x = "global"
def foo():
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

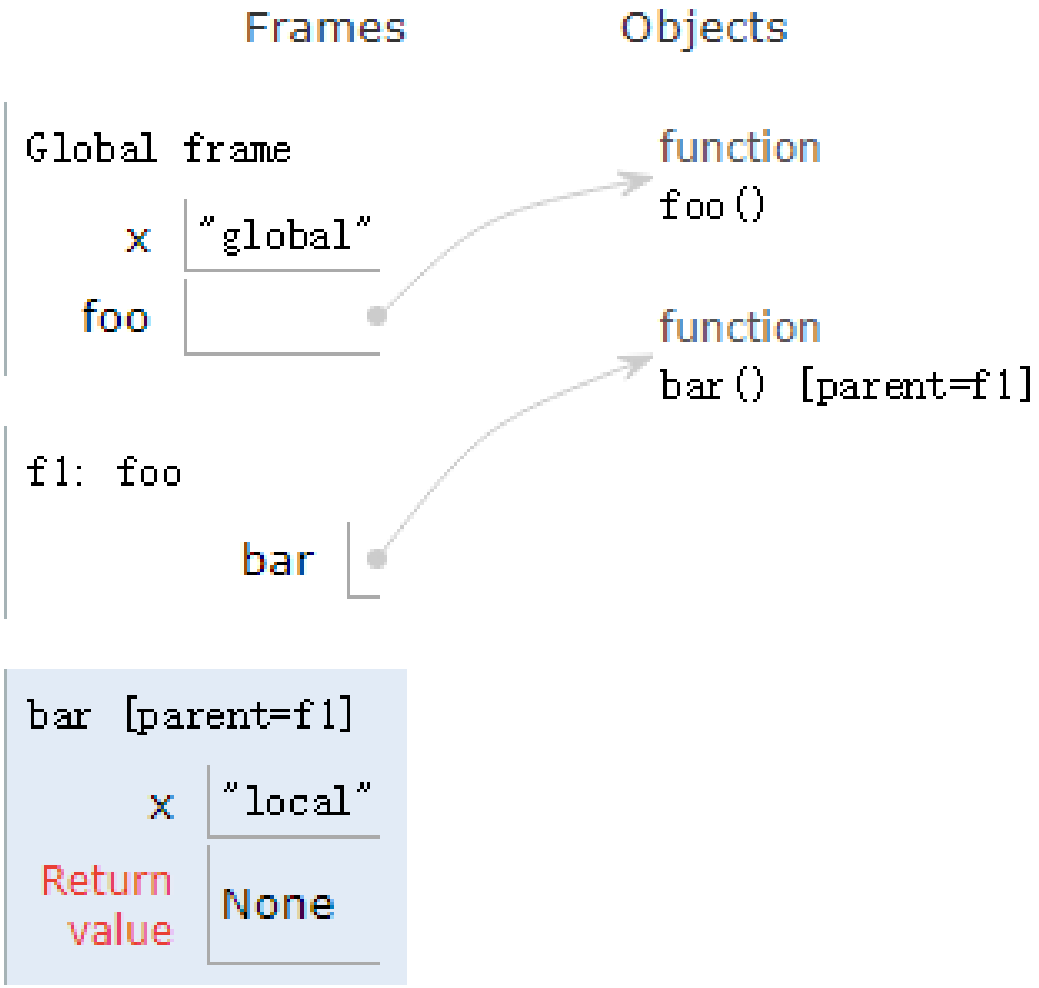
- global x: global – bar
- local x: bar

Output

```
global
global
local
>>>
```

Scope

```
x = "global"
def foo():
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```



print(x) in foo uses global x, not local x

Scope

```
def foo():
    def bar():
        print(x)
        x = "local"
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

- local x: bar

When a name is not found at all, a NameError exception is raised

Scope

```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```

Scopes:

- global
- foo
- bar
- baz

The scope of x

- global x: global
- non-local-foo x: foo
- local-var x: bar
- local-baz x: baz

Scope foo and baz are non-overlapped

Scope

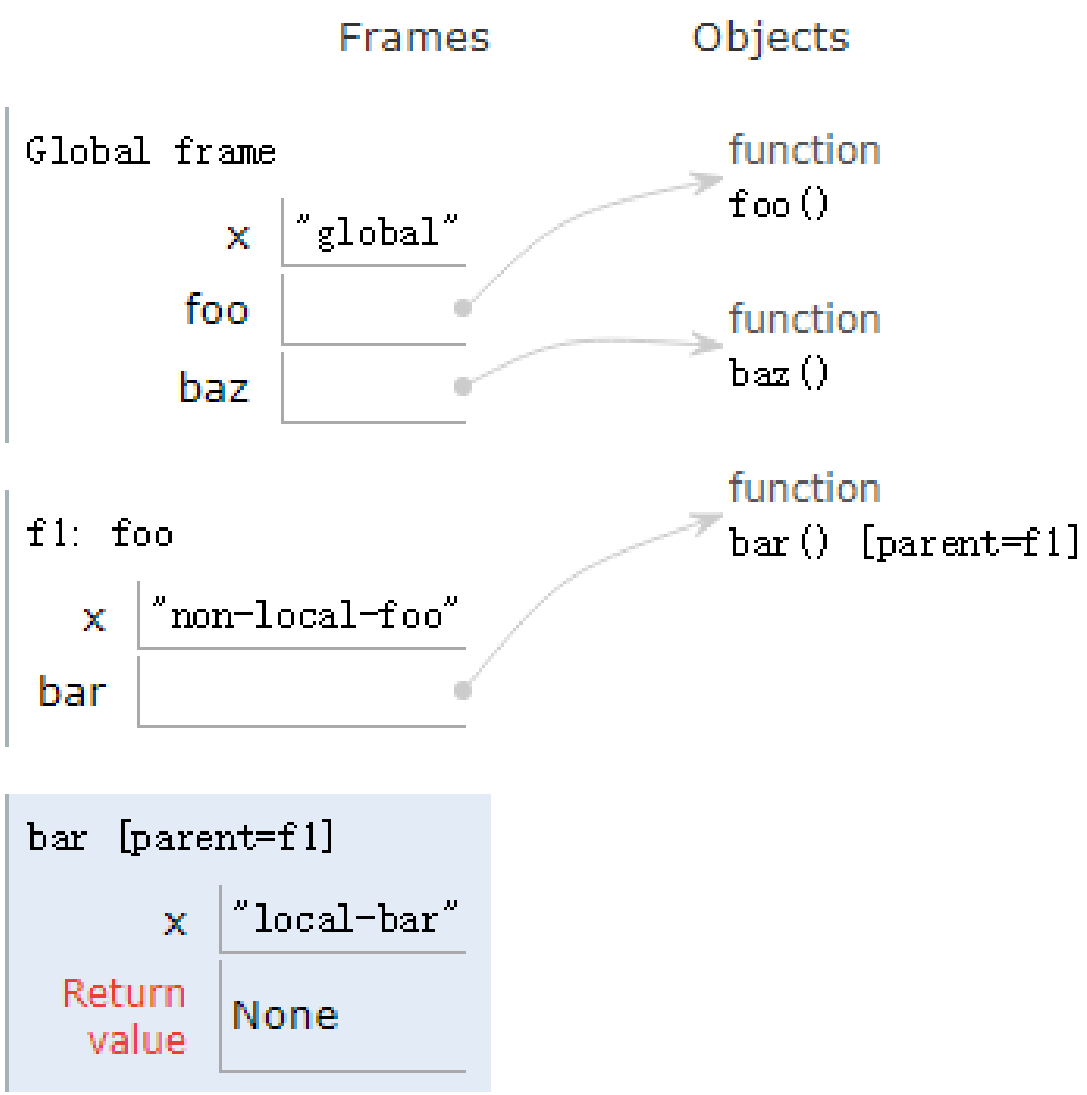
```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```

Output

```
global
non-local-foo
local-bar
local-baz
Traceback (most recent call last):
  File "D:\Test\fib.py", line 16, in
<module>
    bar()
NameError: name 'bar' is not
defined
```

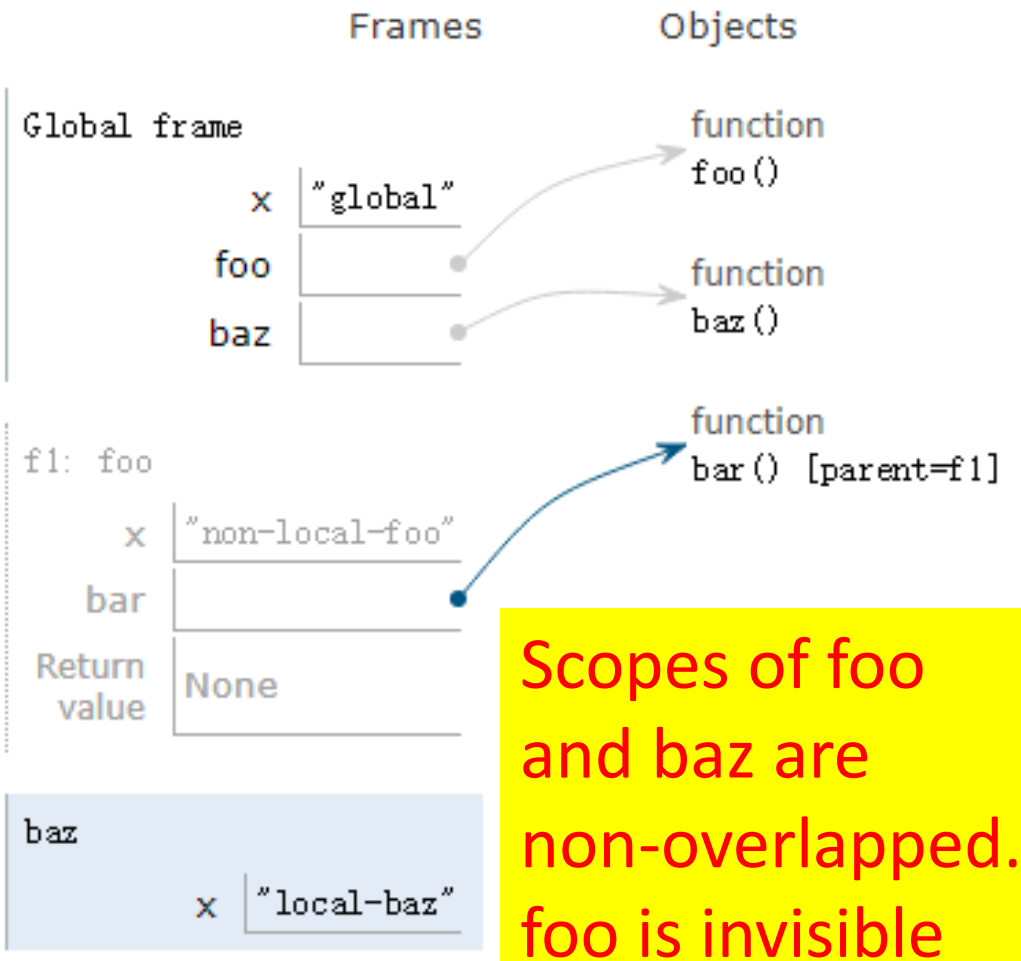
Scope

```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```



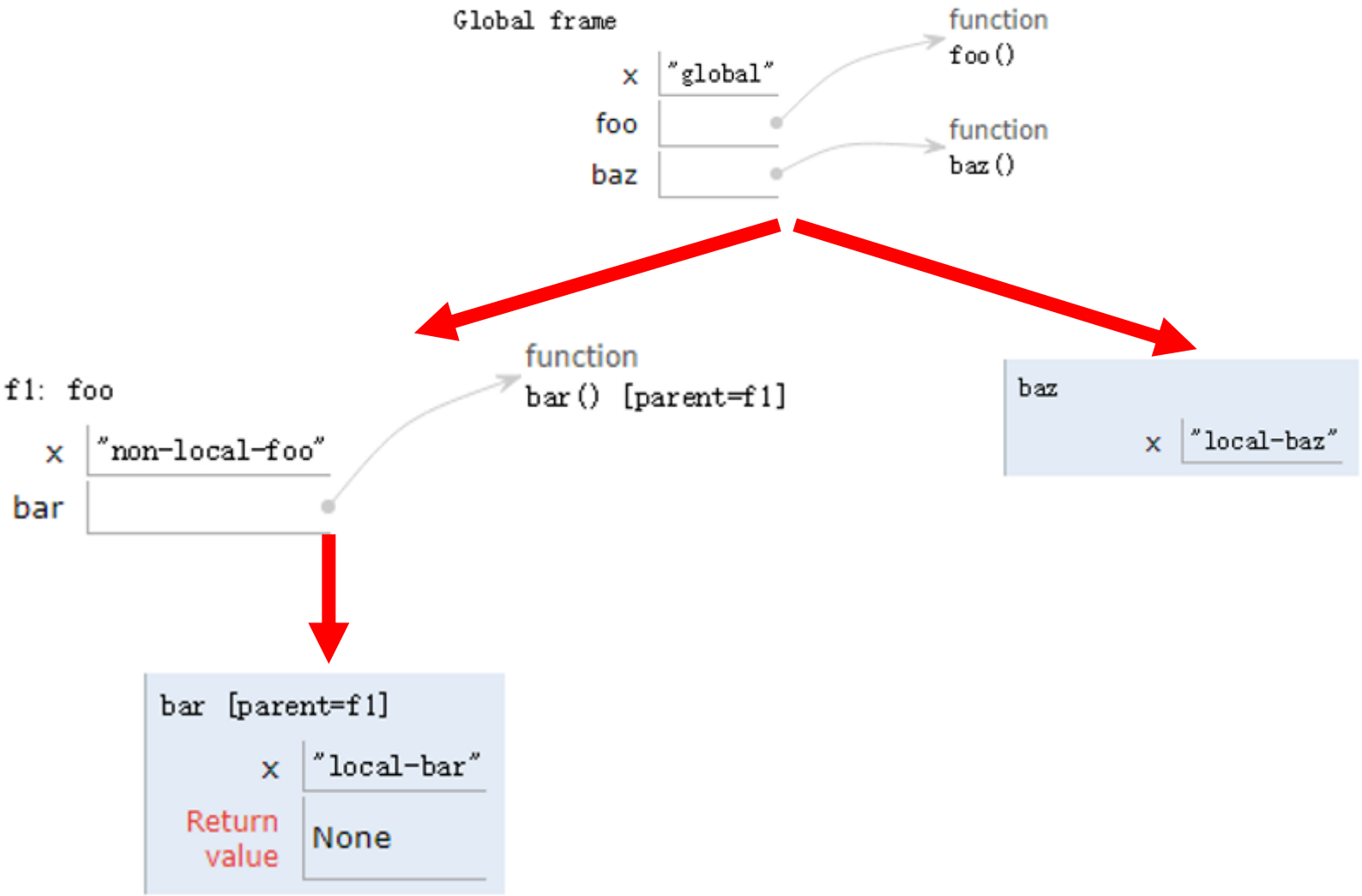
Scope

```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```



Scopes of foo and baz are non-overlapped. foo is invisible in baz

Namespace in tree-structure



Warning

- Scopes are determined **statically**
- If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block
- This can lead to errors when a name is used within a block before it is bound

Use before binding

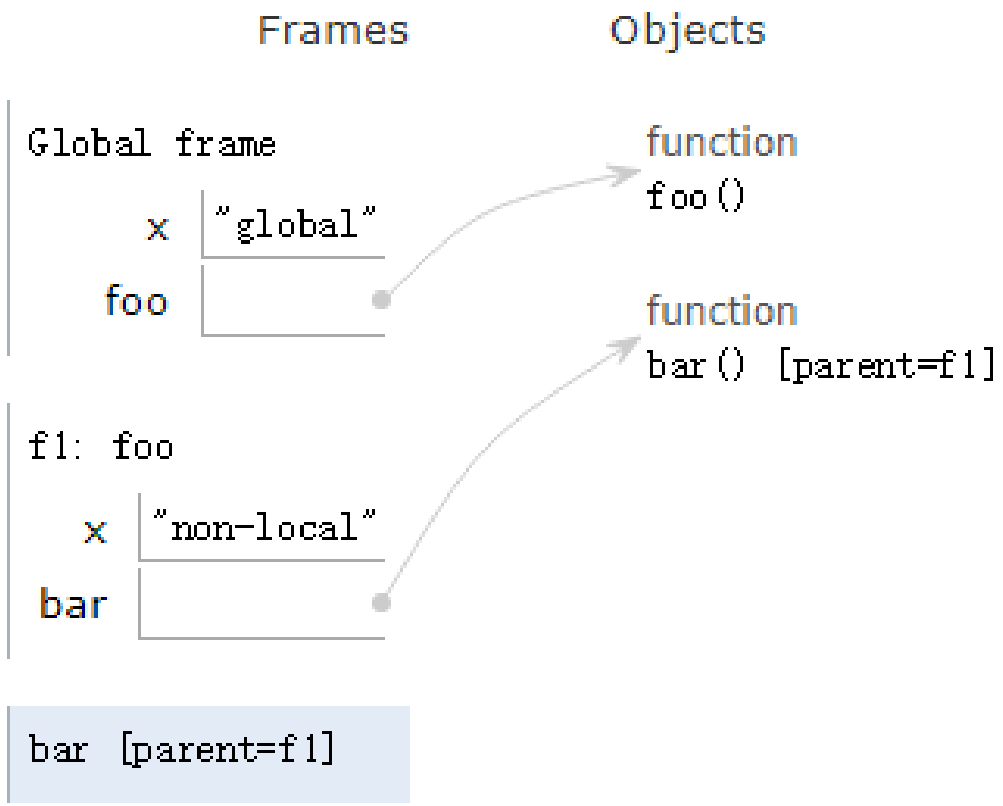
```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
        x = "local"
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
Traceback (most recent call last):
  File "D:\Test\fib.py", line 10, in
<module>
    foo()
  File "D:\Test\fib.py", line 8, in foo
    bar()
  File "D:\Test\fib.py", line 5, in bar
    print(x)
UnboundLocalError: local variable 'x'
referenced before assignment
>>>
```

Use before binding

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
        x = "local"
    print(x)
    bar()
print(x)
foo()
```



x is defined in the local block, but it has not been bound to an object when it is used

Global Statements

- If the global statement global x occurs within a block, all uses of the name x specified in the statement refer to the binding of that name in the global namespace
- The global statement has the same scope as a name binding operation in the same block
- If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global
- If no global x exists, then NameError

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        global x
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
global
>>>
```

Read global x in local block

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        global x
        x = "newglobal"
        print(x)
    print(x)
    bar()
print(x)
foo()
print(x)
```

Output

```
global
non-local
new-global
new-global
>>>
```

Rebinding global x in local block

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        def baz():
            print(x)
        baz()
        global x
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
global
>>>
```

Read global x in nearest enclosing scope, even global is declared after baz()

Warning

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        global x
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
global
```

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
        global x
    print(x)
    bar()
print(x)
foo()
```

SyntaxError: name 'x' is used prior to global declaration (<string>, line 6)

Warning

```
def foo():
    x = "non-local"
    def bar():
        def baz():
            print(x)
        baz()
        global x
    print(x)
    bar()
print(x)
foo()
```

Traceback (most recent call last):
  File "D:\Test\fib.py", line 10, in <module>
    print(x)
NameError: name 'x' is not defined

If no global x exists, then NameError

Nonlocal Statements

- The nonlocal statement nonlocal x causes corresponding name x to refer to previously bound variables in the nearest enclosing function scope (excluding global scope) 不包含global！！
- SyntaxError is raised at compile time if the given name does not exist in any enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        x = "local"
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

```
local
>>>
```

x to refer to previously bound variables in the nearest enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

```
non-local
>>>
```

x to refer to previously bound variables in the nearest enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    def bar():
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

SyntaxError: no binding for nonlocal 'x' found (<string>, line 5)

x cannot refer to previously bound variables in the global scope

Recap

- **Exception**
- **Function**
  - Function definition
  - Function invocation
- **Scope**
  - Local
  - Non-local
  - Global
  - Name resolution