Introduction to Information Science and Technology

(SI100B Python Part)

Fu Song
School of Information Science and Technology
ShanghaiTech University

Review

1. Compound data
    - Numeric types: int, long, float, complex, Bool
    - Sequence types: list, tuple, range, str
    - Mapping types: dict
    - Set types: set, frozen set
    - None type: type(None)
2. Input and Output (IO)
    - Input/print, format string
    - Read/write file

Outlines

1. Exceptions
2. Naming, Binding and Scope

Even the best laid plans sometimes go wrong

- You create a shopping list then when you get to Auchan realize you left the list at home
- You want to buy a pair of shoes, but your size is out of stock
- You need to call someone and your cell phone battery is dead

Things go wrong in programs as well

- A program cannot find a file it needs
- A user enters a date in the wrong format
- You try to divide a number by zero

Exceptions

```
>>> f = open("D://test2.txt", 'r+')
Traceback (most recent call last):
  File "<pyshell#427>", line 1, in <module>
    f = open("D://test2.txt", 'r+')
FileNotFoundError: [Errno 2] No such file or directory: 'D://test2.txt'
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#428>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>>
```

How can we handle such kind of error so that the program can continue its execution???

- Exception handling: is the process of responding to the occurrence, during computation, of exceptions
- Most of high-level programming languages provide mechanism to handle exceptions, like Java, C#, Python

Handling Exceptions

Exception

* built-in Exception
* custom Exception
* raise exception
* handling Exception

Form

```
try:
      statements
except  Error1:
      statements
except  Error2:
      statements
  ......
else:   # optional
      statements
finally: # optional
      statements
```

Handling Exceptions

Test.py

```
try:
    f = open("D://test2.txt", 'r')
except FileNotFoundError:
    print("D://test2.txt does not exist")
print("Still executing")
```

```
D://test2.txt does not exist
Still executing
>>>
```

Handling Exceptions

$$Area\ \Delta ABC = \sqrt{s(s-a)(s-b)(s-c)} \quad where\ \ s = \frac{a+b+c}{2}$$

```
>>> import math
>>> def Area(a, b, c):
        if (a + b <=c) or (a + c <=b) or (b + c <=a):
                print("It is not a triangle")
                return None
        s = (a + b + c) /2
        area = math.sqrt(s * (s-a) * (s-b) * (s-c))
        return area

>>> Area(2,2,5)
It is not a triangle
>>> Area(3,4,5)
6.0
>>>
```

Handling Exceptions

```
import math
def Area(a,b,c):
    s = (a+b+c)/2
    try:
        print("Before Exception")
        area = math.sqrt(s *(s-a)*(s-b)*(s-c))
        print("After Exception")
    except ValueError:
        print("It is not a triangle")
    else:
        print("Succeed")
        return area
    finally:
        print("At finally")

Area(3,4,5)
Area(2,2,5)
```

```
Before Exception
After Exception
Succeed
At finally
Before Exception
It is not a triangle
At finally
>>>
```

Handling Exceptions

```
import math
def Area(a,b,c):
    s = (a+b+c)/2
    try:
        print("Before Exception")
        area = math.sqrt(s *(s-a)*(s-b)*(s-c))
        print("After Exception")
    except ZeroDivisionError:
        print("It is not a triangle")
    else:
        print("Succeed")
        return area
    finally:
        print("At finally")

Area(2,2,5)
```

```
Before Exception
At finally
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
  File "C:\Program Files\Python\lib\idlelib\run.py", line 353, in runcode
    exec(code, self.locals)
  File "C:/Users/Fu Song/Desktop/area.py", line 16, in <module>
    Area(2,2,5)
  File "C:/Users/Fu Song/Desktop/area.py", line 6, in Area
    area = math.sqrt(s *(s-a)*(s-b)*(s-c))
ValueError: math domain error
```

Built-in Exceptions

Common built-in exceptions

- ZeroDivisionError: division by zero
- ValueError: value does not meet condition
- KeyError: key does not exist in a dict
- TypeError:

Details refer to
"The Python Standard Library" section 5. Built-in Exceptions

Built-in Exceptions

```
>>> d = {1:1,2:2}
>>> d[3]
Traceback (most recent call last):
  File "<pyshell#481>", line 1, in <module>
    d[3]
KeyError: 3
```

```
>>> 'ab'-2
Traceback (most recent call last):
  File "<pyshell#485>", line 1, in <module>
    'ab'-2
TypeError: unsupported operand type(s) for -: 'str' and 'int'
>>>
```

Nested Exceptions

```
try:
    ...... # Outer try block
    try:
        ...... # Inner try block
    except Error1:
        ...... # Inner exception handler
    ...... # End of the outer try block
except Error2:
    ...... #  Outer exception handler
```

Search matched except catch:

- first current `try except` block
- if not matched except is found,
  - ✓ execute finally block of the current block, then
  - ✓ go outer side block until catched
- `unhandled exception`

User-defined Exceptions

```
class MyZeroDivisionError(Exception):
    def __init__(self, msg):
        self._msg = msg
```

```
try:
    f(1, 1, 1)
except MyZeroDivisionError as e:
    print(e._msg)
```

```
def f(a, b, c):
    if a==b and b==c:
        raise MyZeroDivisionError("MyError")
```

```
MyError
>>>
```

Open file with Exception handler

test.py - C:/Users/Fu Song/De

File Edit Format Run Options Window Help

```
fn1 = input("Input file name: ")
fn2 = input("Input file name: ")
try:
    f1 = open(fn1, 'w')
    f2 = open(fn2, 'r')
    content = f2.read()
    f1.write(content)
except FileNotFoundError:
    print("{0} does not exist!".format(fn2))
else:
    print("{0} does exist!".format(fn2))
    f2.close()
finally:
    f1.close()
```

- **Programs should always open a file, and close it when they are done**
- **If they don't sometimes the code crashes when you try to re-open a file that wasn't closed last time you ran your code**

Open file with Exception handler

```
fn1 = input("Input file name: ")
fn2 = input("Input file name: ")
with open(fn1, 'w') as f1:
    with open(fn2, 'r') as f2:
        content = f2.read()
        f1.write(content)
```

- The 'with' ':' syntax is used for certain methods to make sure clean up code such as close file runs even if there is an error.

Outlines

1. Exceptions
2. Naming, Binding, Scope
3. Classes

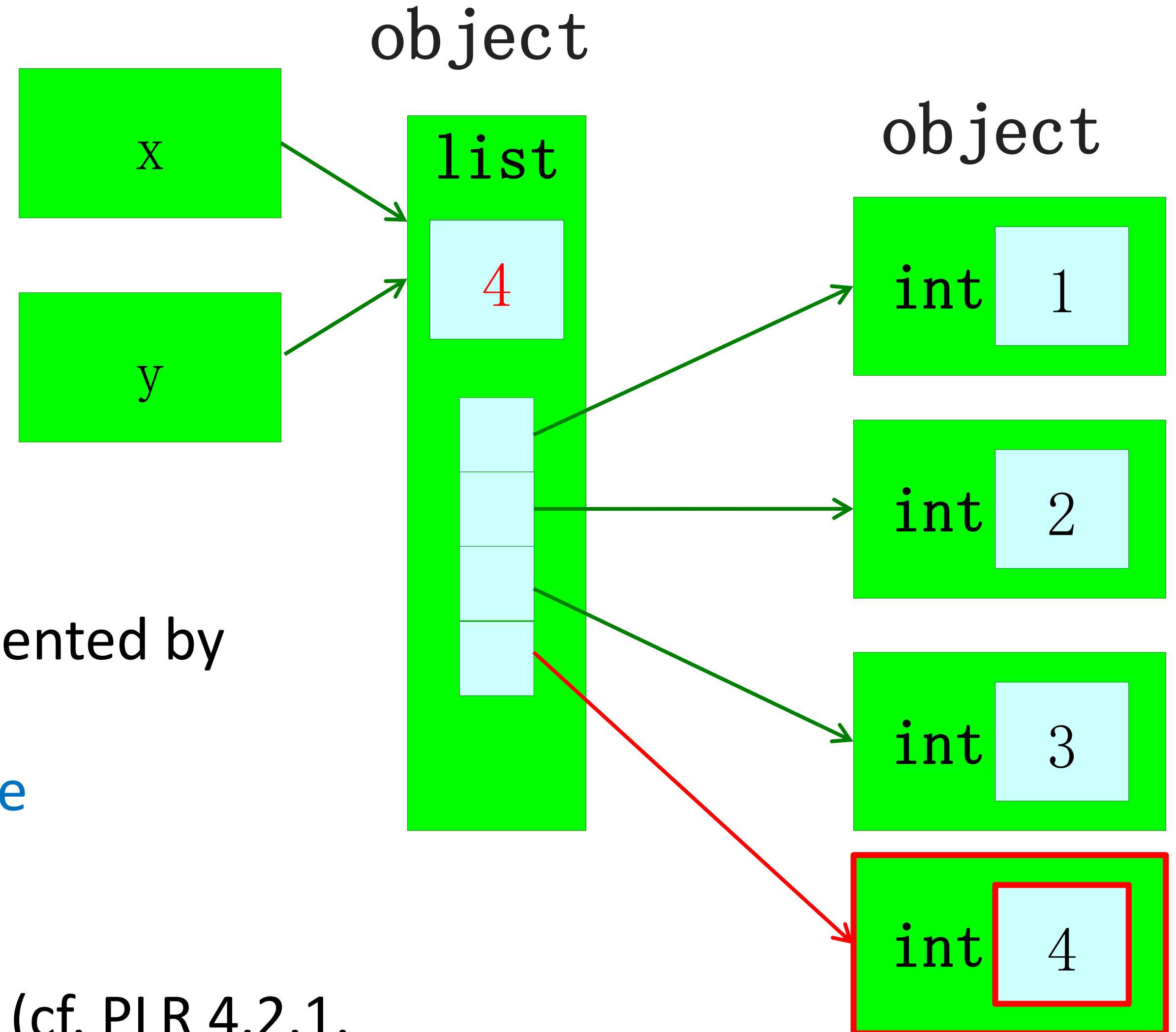
Naming and Binding

```
x = [1,2,3]
y = x
x.append(4)
```

x is ?
y is ?
Why?

Naming and Binding

```
x = [1,2,3]
y = x
x.append(4)
```



Object

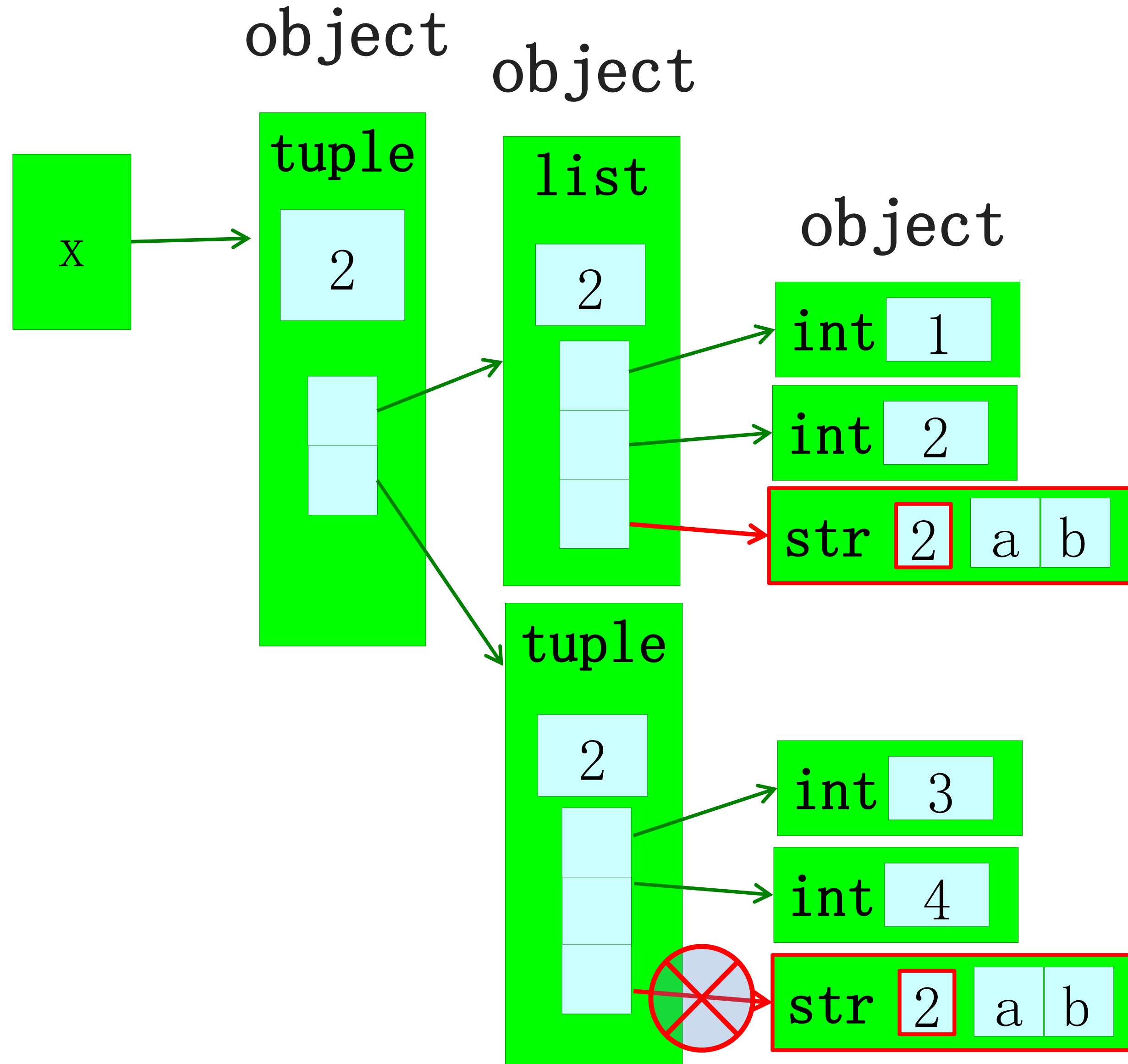
- each data in a Python program is represented by objects or by relations between objects
- object has an identity, a type and a value
- id(x), type(x)

Name

- introduced by name binding operations (cf. PLR 4.2.1.
- refers to objects

Mutable vs. Immutable

```
x = ([1,2],(3,4))
x[0].append('ab')
x[1].append('ab')
```



Scope and Namespace

```
def foo():
    x = "local"
    print("Inside of foo: {0}".format(x))

x = "global"
foo()
print("Outside of foo: {0}".format(x))
```

Which object does x refer to?

Scope and Namespace

Namespace: a map from names to objects

Block: a piece of Python program text that is executed as a unit

- module
- function definition
- class definition
- Script file

Each block has its own namespace

Scope and Namespace

Namespace: a map from names to objects

Block: a piece of Python program text that is executed as a unit

- module
- function definition
- class definition
- Script file

Each block has its own namespace

```
def foo():
    x = "local"
    print("Inside of foo: {0}".format(x))

x = "global"
foo()
print("Outside of foo: {0}".format(x))
```

- foo Namespace: x inside foo
- file Namespace: x outside of foo

Scope and Namespace

- Local: name defined in a block is a local variable of this block
- Global: name defined in the module block is a global variable of this block
  - ✓ The variables of the module code block are local and global
- Non-local: names defined between local and global scopes.
- Built-in: built-in names

File Edit Format Run Optio

```
def f():
    x = "nonlocal"
    def g():
        x = "local"
        print(x)
    g()
x = "global"
f()
```

>>> local

Scope and Namespace

Scope: defines the visibility of a name within a block.

- If a local variable 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.

How are names resolved?

Without nonlocal or global modifier:

- When binding a name, place it in the local scope.

  E.g., assignment, formal parameters, import, function and class definitions

- When using a name, search
  1. local
  2. non-local
  3. global
  4. built-in

```
def f():
    x = "nonlocal"
    def g():
        x = "local"
        print(x)
    print(x)
    g()
x = "global"
print(x)
f()
```

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

```
def f():
    x = "nonlocal"
    def g():
        # x = "local"
        print(x)
    print(x)
    g()
x = "global"
print(x)
f()
```

```
global
nonlocal
nonlocal
>>>
```

```
def f():
    #x = "nonlocal"
    def g():
        # x = "local"
        print(x)
    print(x)
    g()
x = "global"
print(x)
f()
```

```
global
global
global
>>>
```

How are names resolved?

With nonlocal or global modifier:

- use nonlocal or global namespace

```
def f():
    x = "nonlocal"
    def g():
        # x = "local"
        print(x)
    print(x)
    g()
x = "global"
print(x)
f()
```

```
global
nonlocal
nonlocal
>>>
```

```
def f():
    x = "nonlocal"
    def g():
        global x
        print(x)
    print(x)
    g()
x = "global"
print(x)
f()
```

```
global
nonlocal
global
>>>
```

UnboundLocalError and NameError

1. NameError
   - When a name is not found at all, a NameError exception is raised.
2. UnboundLocalError
   - If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an UnboundLocalError exception is raised
   - UnboundLocalError is a subclass of NameError

```
def f():
    # x = "nonlocal"
    def g():
        print(x)
    g()
#x = "global"
#print(x)
f()
```

```
def f():
    # x = "nonlocal"
    def g():
        x = "local"
        print(x)
    g()
    print(x)
#x = "global"
#print(x)
f()
```

NameError

```
def f():
    x = "nonlocal"
    def g():
        print(x)
        x = "local"
    print(x)
    g()
x = "global"
print(x)
f()
```

UnboundLocalError

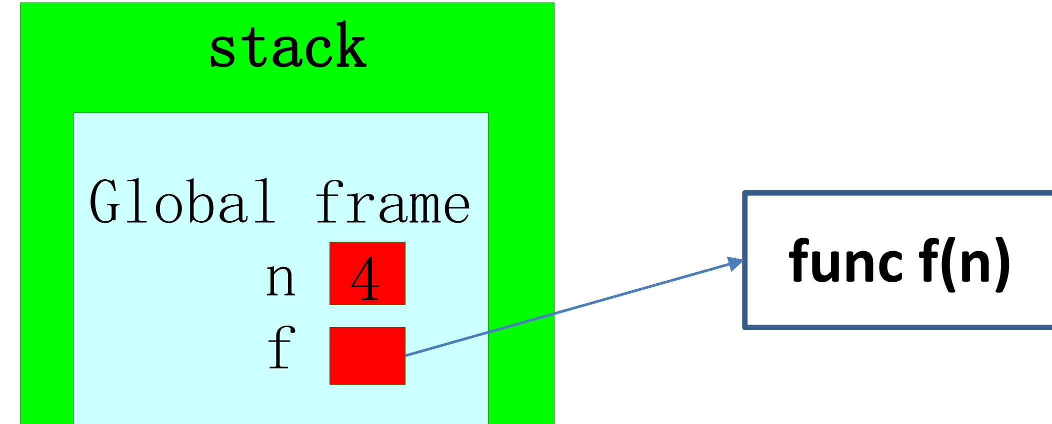
Execution Frame

- Scopes are determined statically, used dynamically
- A code block is executed in an execution frame
  - ✓ contains some administrative information (used for debugging)
  - ✓ determines where and how execution continues after the code block's execution has completed

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```

Layout of Frames



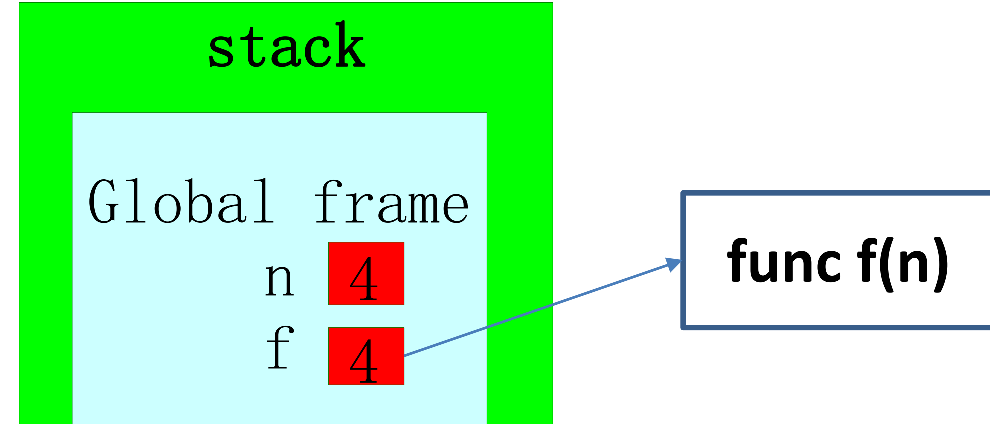

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)
```

```
n = 4
f(4)
```

Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**




```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)
```

```
n = 4
f(4)
```

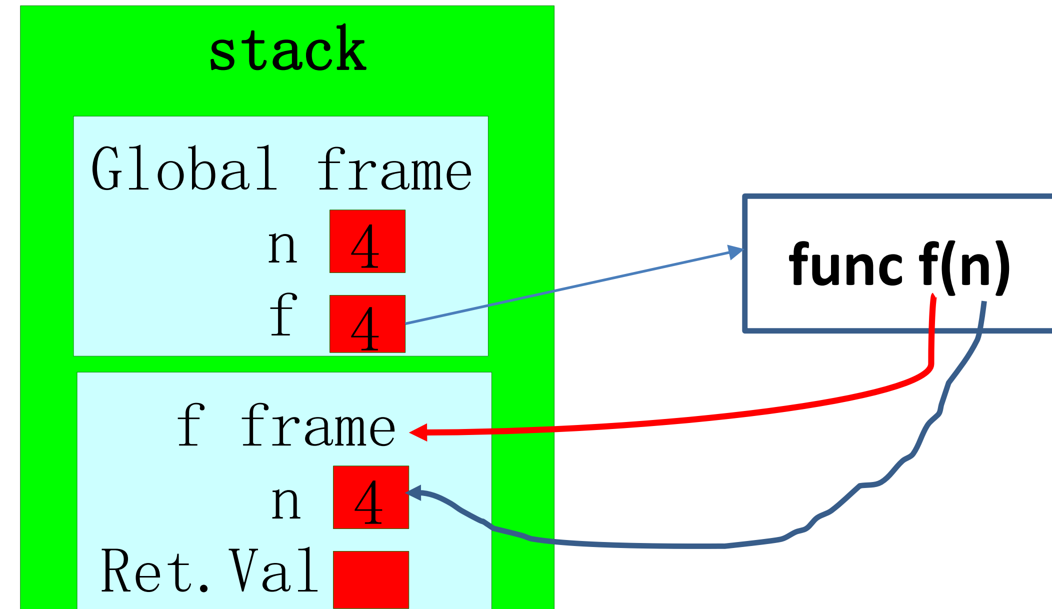
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)
```

```
n = 4
f(4)
```



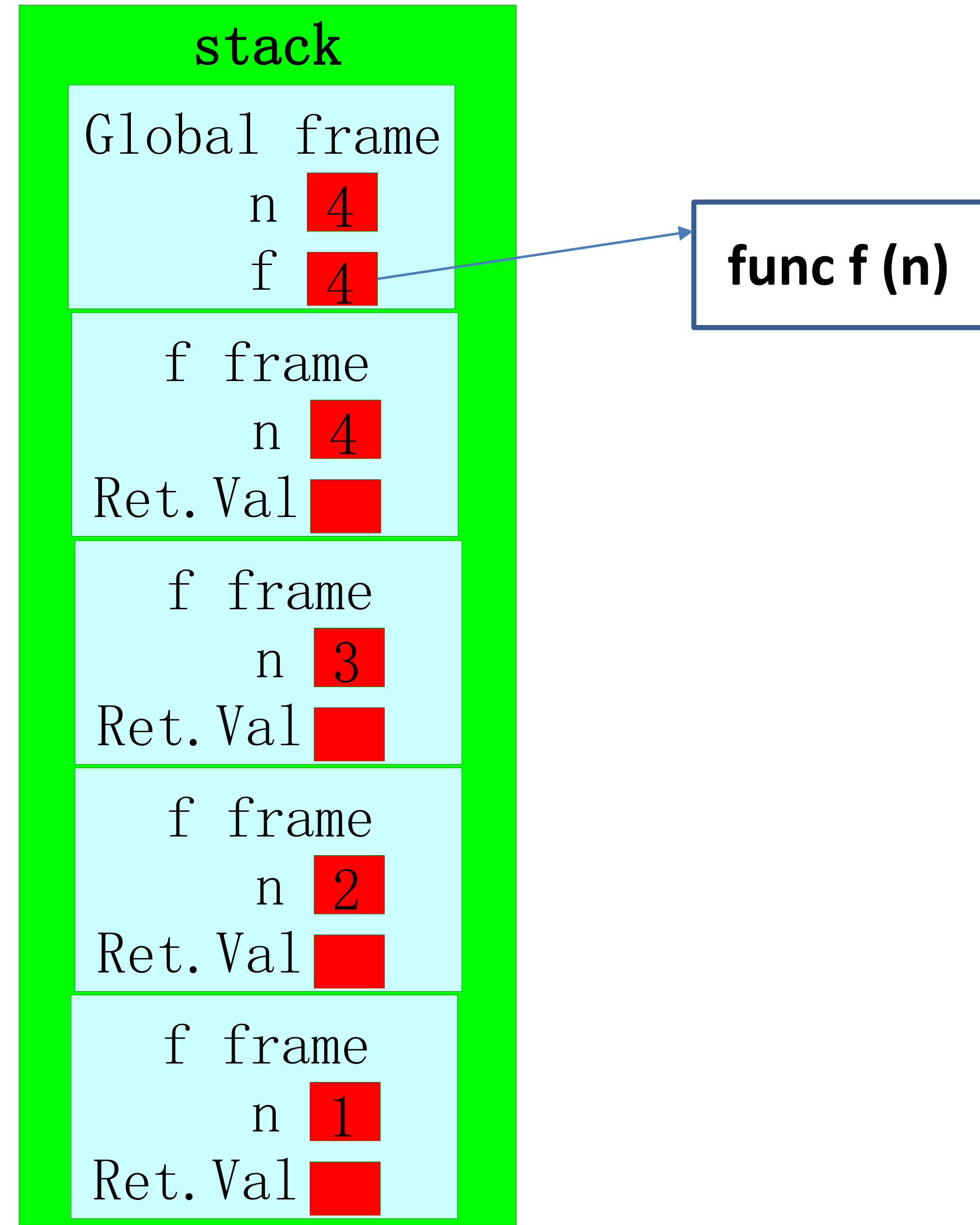
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



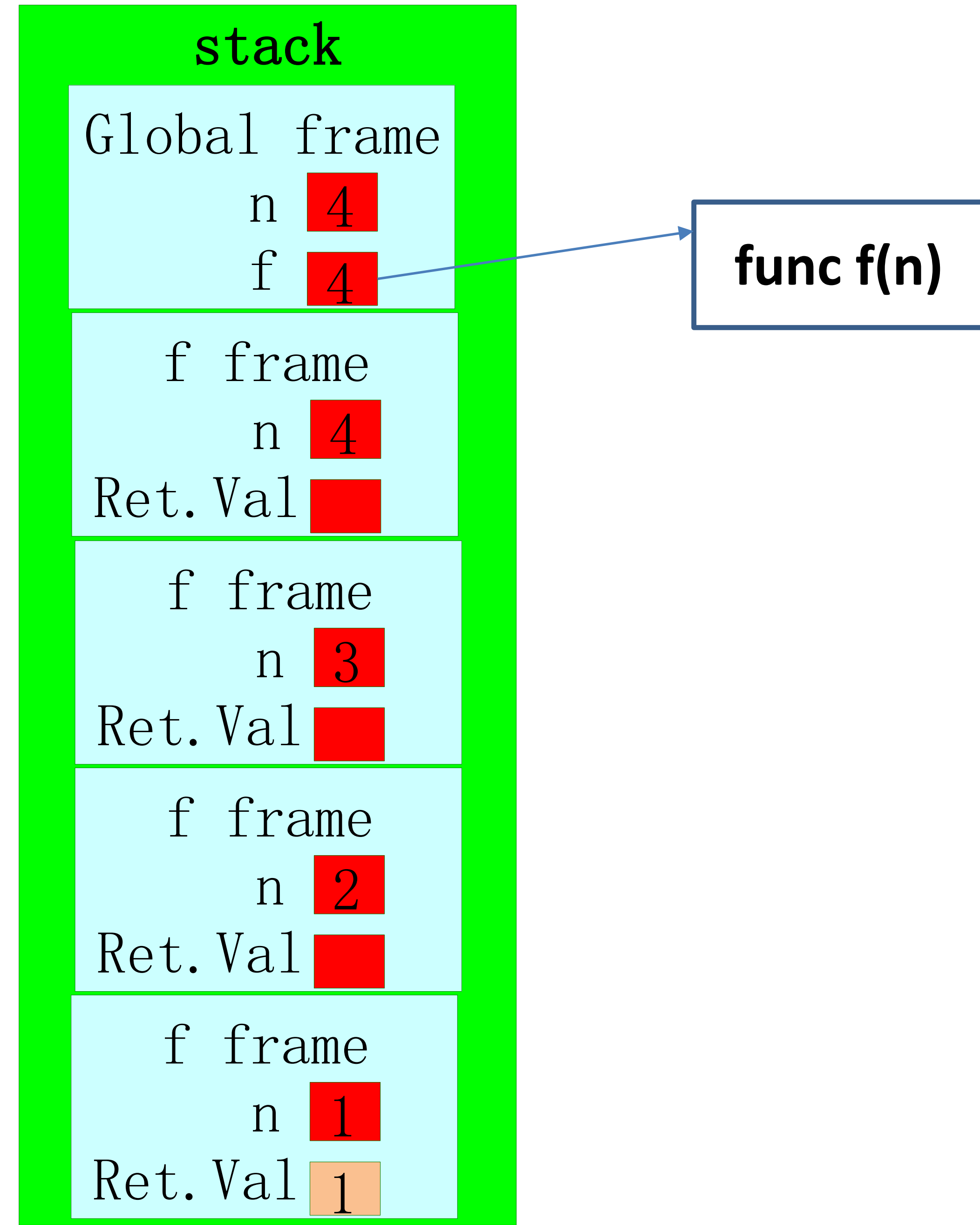
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



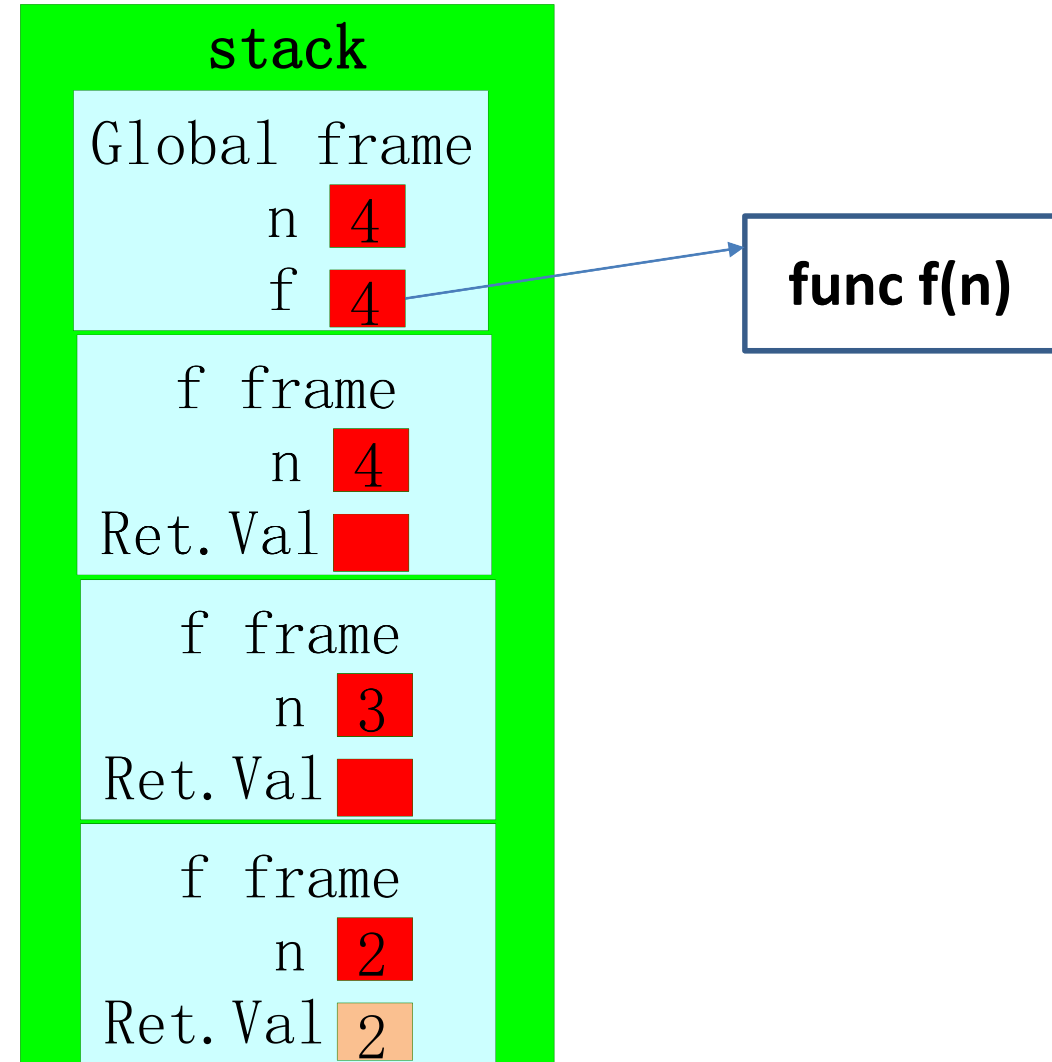
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    '''  Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



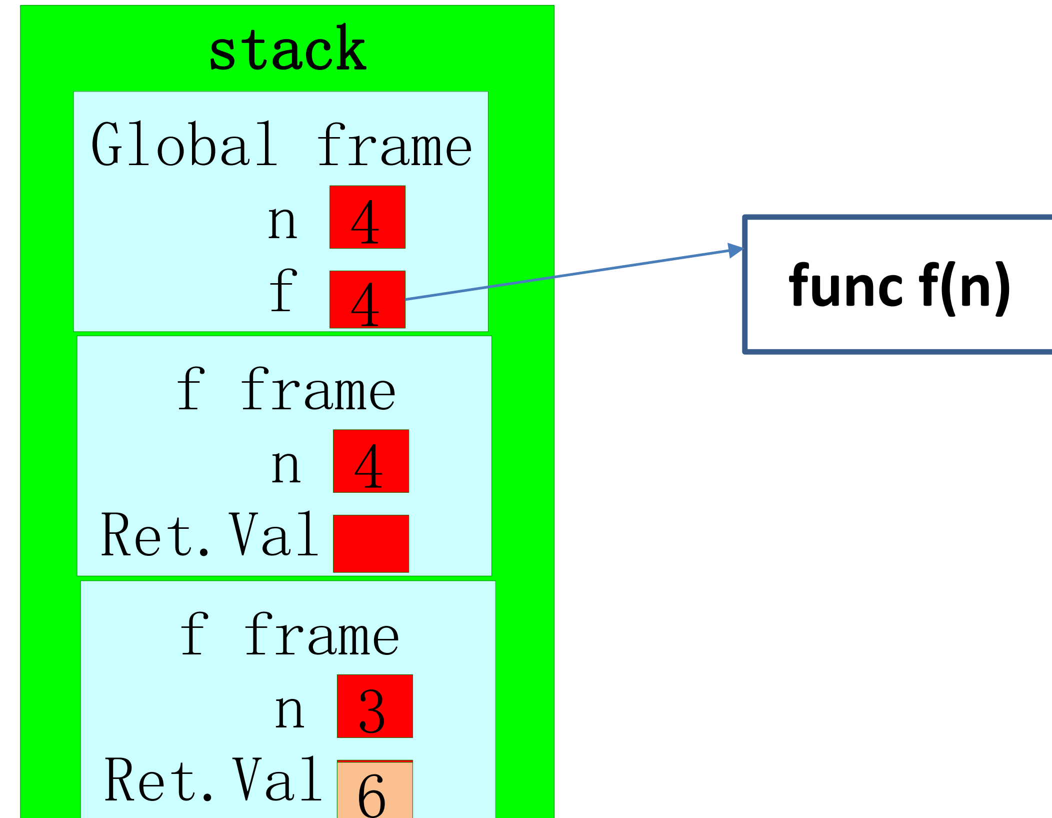
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



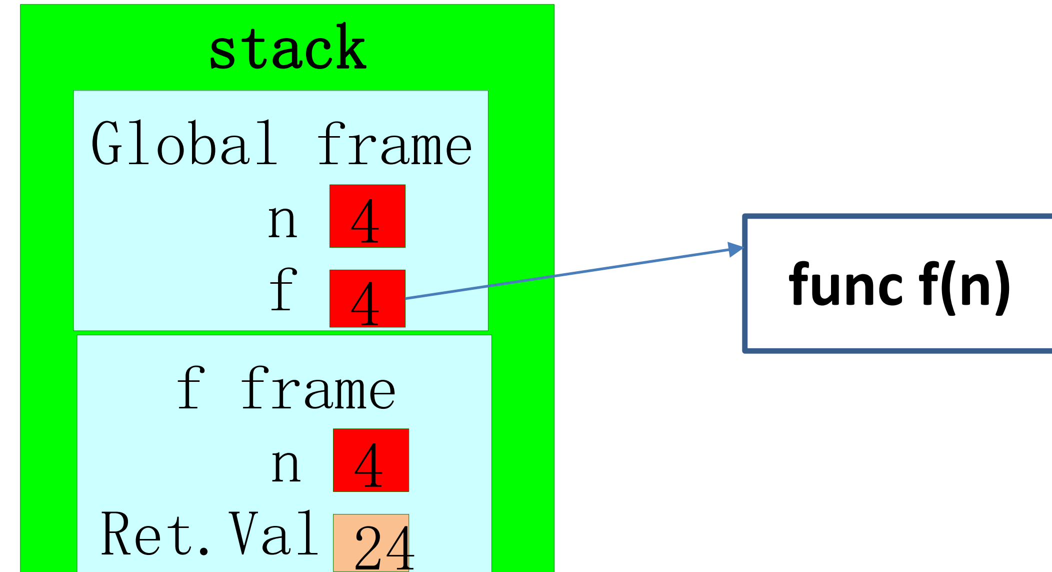
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



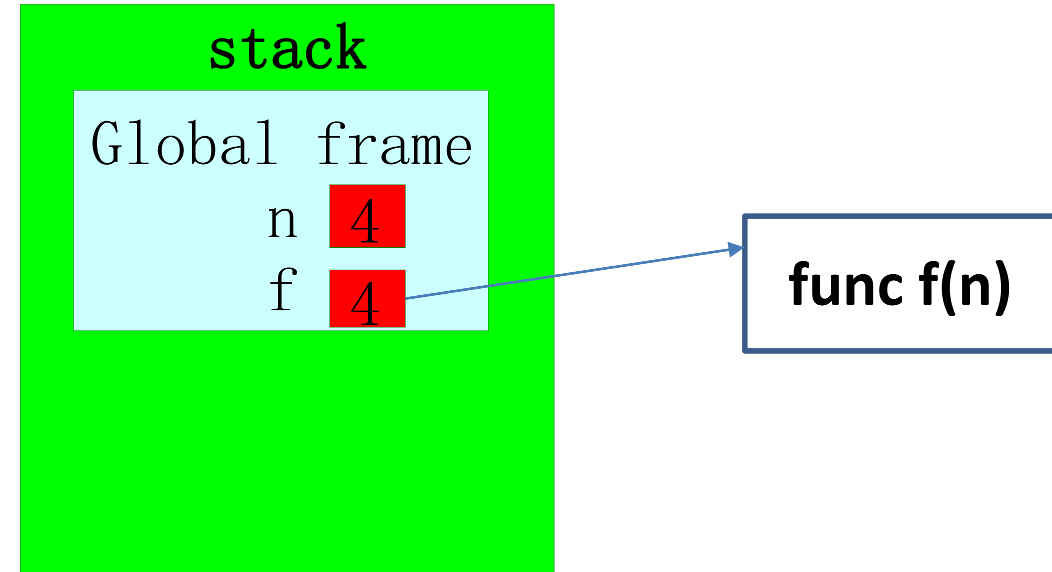
Layout of Frames

Procedure for calling/applying user-defined functions

1. **Add a local frame**
2. **Bind the function's formal parameters to its arguments in that frame**
3. **Execute the body of the function in that new frame**

```
def f(n):
    ''' Input: n>1
    Output: n!'''
    if n == 1:
        return 1
    else:
        return n * f(n - 1)

n = 4
f(4)
```



Quiz

Read the following program:

```
def foo(a,b):
    s =''
    try:
        x= a/b
        s='Succeed'
    except ZeroDivisionError:
        s = s+'ZeroDivisionError'
    else: s = s + 'Else'
    finally: s= s + 'Finally'
    return s
foo(1,0)+foo(1,1)
```

What does the above program output?

ZeroDivisionErrorFinallySucceedElseFinally

Read the following program:

```
x = 4
def f ():
    x = 2
    def g():
        x = 3
        return lambda z: x*z
    return lambda z:g()(x*z)
f()(x)
```

What does the above program output?