

Python程序设计

函数 - 基础知识与递归

刘安

苏州大学，计算机科学与技术学院

<http://web.suda.edu.cn/anliu/>

本节涉及到的知识点

- 函数的定义和使用
- 变量的作用域和参数传递
- 高阶函数
- 函数嵌套
- 递归

创建和使用函数

- def语句创建一个函数对象并将其赋值给一个变量

```
1 def intersect(A, B):  
2     res = []  
3     for x in A:  
4         if x in B:  
5             res.append(x)  
6     return res
```

- 执行def语句之后，创建了一个函数对象，并将变量intersect绑定到该对象 **简单起见，直接称呼函数intersect**
- 在交互模式中使用函数：函数名(参数)

```
>>> C = intersect(LA, LB)
```



def也是复合语句，所以首行末尾需要冒号，其后的代码块需要缩进

函数调用

- 当函数调用时，调用者暂停执行，直到该函数遇到`return exp`语句执行结束，并返回表达式`exp`的值

```
1 def intersect(A, B):  
2     res = []  
3     for x in A:  
4         if x in B:  
5             res.append(x)  
6     return res
```

- 调用者获得该返回值，并继续程序的执行

```
>>> LA = [1, 2, 3, 4, 5]  
>>> LB = [2, 3, 5, 7, 11]  
>>> C = intersect(LA, LB)  
>>> |
```



如何使用列表推导计算两个列表的交集？

return语句

- return语句结束函数调用，并且：
 - 如果return语句包含一个表达式，则返回该表达式的值
 - 如果return语句不包含表达式，则返回一个None对象
- 如果函数没有return语句，则执行完其内部所有语句后结束函数调用，并自动返回一个None对象

```
>>> def g():  
    x = 1  
    return x
```

```
>>> print(g())  
1
```

```
>>> def h():  
    x = 1  
    if x: return  
    return x
```

```
>>> print(h())  
None
```

```
>>> def f():  
    x = 1
```

```
>>> print(f())  
None
```

考拉兹猜想

- 对于任何一个正整数，如果它是奇数，对它乘3再加1，如果它是偶数，对它除以2，如此循环，最终都能够得到1

考拉兹序列 13 40 20 10 5 16 8 4 2 1

- 编写一个函数collatz，具有一个名为n的参数
 - 如果n是奇数，返回 $3*n+1$
 - 如果n是偶数，返回 $n//2$
- 编写一个函数trace_collatz，接受一个正整数，以列表形式返回其对应的考拉兹序列

考拉兹猜想

```
1 def collatz(n):
2     if n % 2 == 1:
3         return 3 * n + 1
4     else:
5         return n // 2
6
7 def trace_collatz(n):
8     res = [n]
9     while True:
10        n = collatz(n)
11        res.append(n)
12        if n == 1:
13            break
14    return res
```

```
>>> trace_collatz(1)
[1, 4, 2, 1]
>>> trace_collatz(2)
[2, 1]
>>> trace_collatz(3)
[3, 10, 5, 16, 8, 4, 2, 1]
```

变量的作用域

- 在函数内部赋值的变量只能在函数内部使用

```
>>> def intersect(A, B):  
    res = []  
    for x in A:  
        if x in B:  
            res.append(x)  
    return res
```

```
>>> res
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#16>", line 1, in <module>
```

```
    res
```

```
NameError: name 'res' is not defined
```


变量的作用域

- 在函数内部赋值的变量和在函数外部赋值的变量即使同名，也是两个不同的变量

```
>>> def intersect(A, B):  
    res = []  
    for x in A:  
        if x in B:  
            res.append(x)  
    return res
```

这是两个不同的变量

```
>>> res = 'outside variable res'  
>>> intersect([1, 2, 3], [2, 3, 5])  
[2, 3]  
>>> res  
'outside variable res'
```

参数传递

- 调用函数的时候需要提供实际参数
 - 不可变类型 (数值、字符串等) : 形参获得实参的值
 - 可变类型 (列表等) : 形参获得实参的引用

```
>>> def f(a, b):  
    a = 8  
    b[0] = 10  
  
>>> x  
1  
>>> y  
[10, 3, 5]  
  
>>> x = 1  
>>> y = [1, 3, 5]  
>>> f(x, y)
```

→ line that has just executed

→ next line to execute

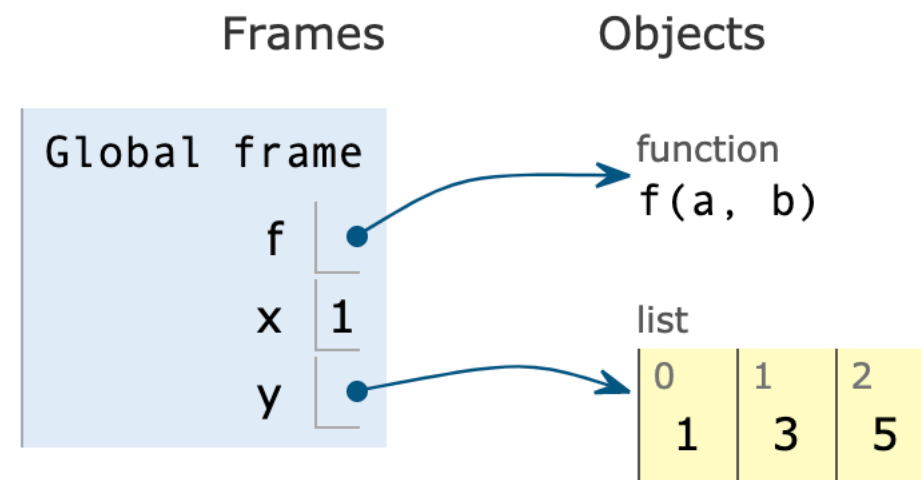
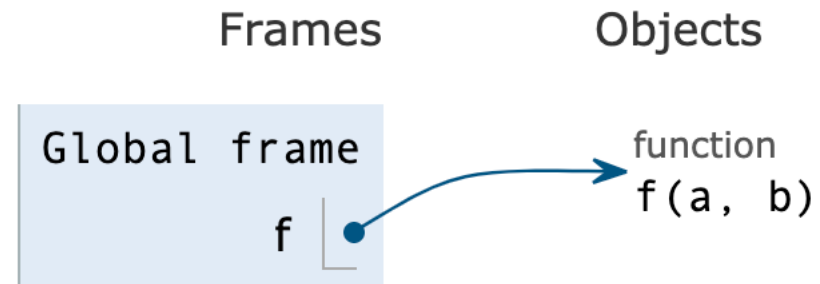
```
→ 1 def f(a, b):  
  2     a = 8  
  3     b[0] = 10  
  4  
  5 x = 1  
  6 y = [1, 3, 5]  
  7 f(x, y)
```

```
→ 1 def f(a, b):  
  2     a = 8  
  3     b[0] = 10  
  4  
→ 5 x = 1  
  6 y = [1, 3, 5]  
  7 f(x, y)
```

```
  1 def f(a, b):  
  2     a = 8  
  3     b[0] = 10  
  4  
  5 x = 1  
→ 6 y = [1, 3, 5]  
→ 7 f(x, y)
```

Frames

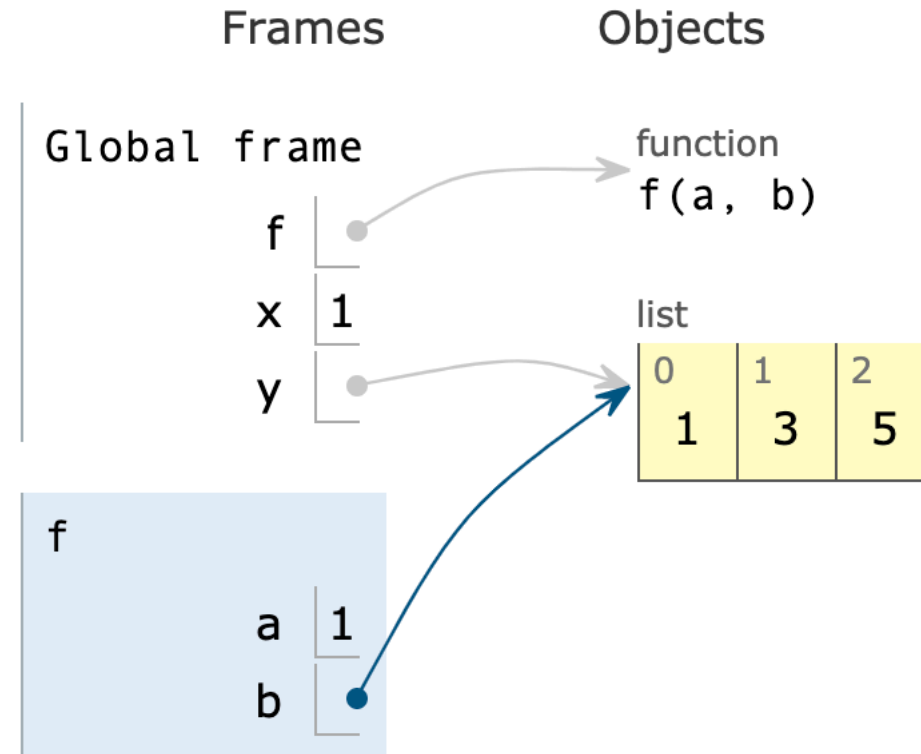
Objects



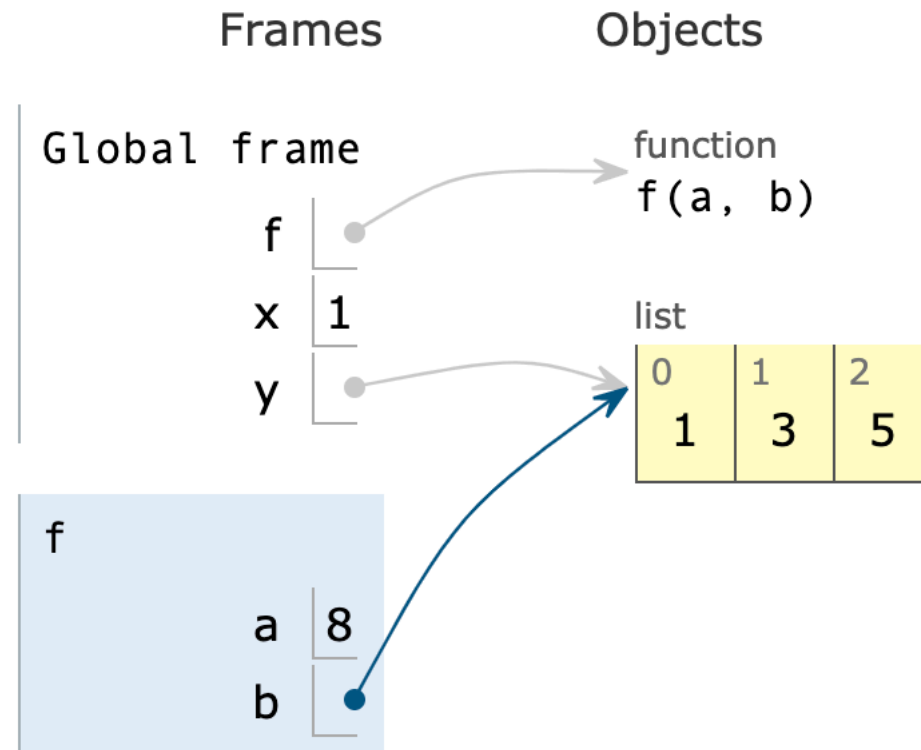
→ line that has just executed

→ next line to execute

```
→ 1 def f(a, b):  
  2     a = 8  
  3     b[0] = 10  
  4  
  5 x = 1  
  6 y = [1, 3, 5]  
→ 7 f(x, y)
```



```
  1 def f(a, b):  
→ 2     a = 8  
→ 3     b[0] = 10  
  4  
  5 x = 1  
  6 y = [1, 3, 5]  
  7 f(x, y)
```

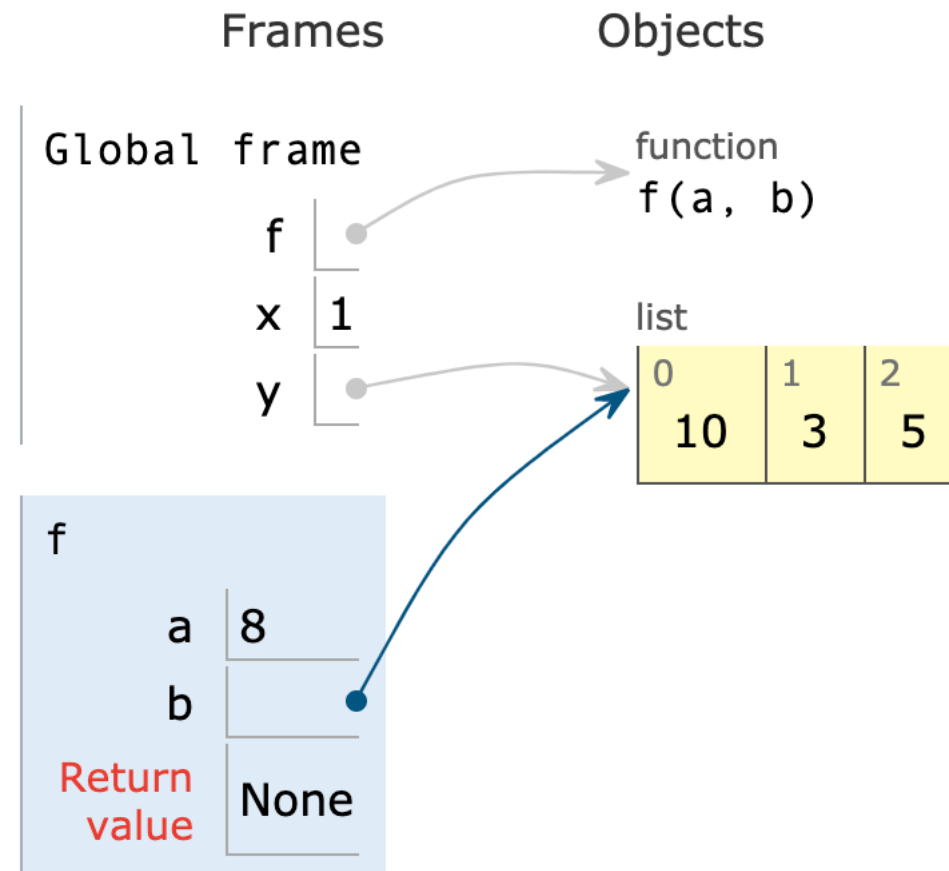


上图通过代码可视化工具生成 - <http://www.pythontutor.com/>

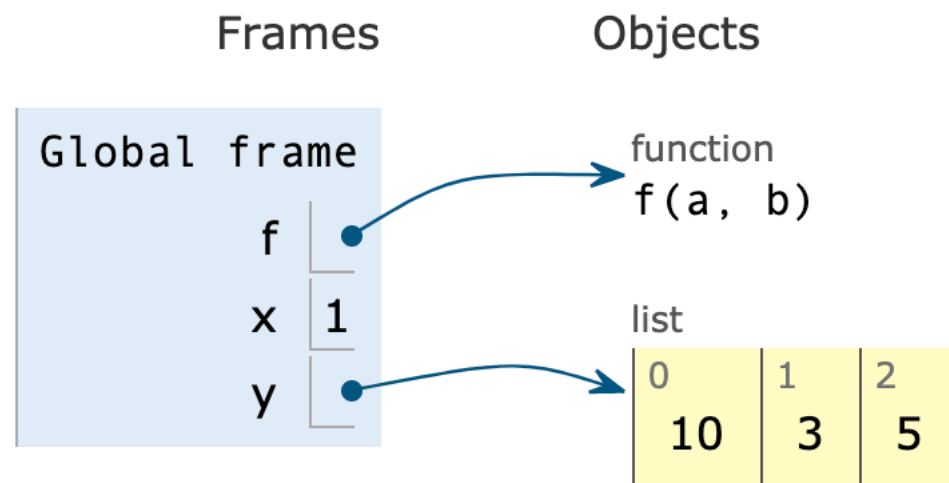
→ line that has just executed

→ next line to execute

```
1 def f(a, b):  
2     a = 8  
→ 3     b[0] = 10  
4  
5 x = 1  
6 y = [1, 3, 5]  
7 f(x, y)
```



```
1 def f(a, b):  
2     a = 8  
3     b[0] = 10  
4  
5 x = 1  
6 y = [1, 3, 5]  
→ 7 f(x, y)
```



上图通过代码可视化工具生成 - <http://www.pythontutor.com/>

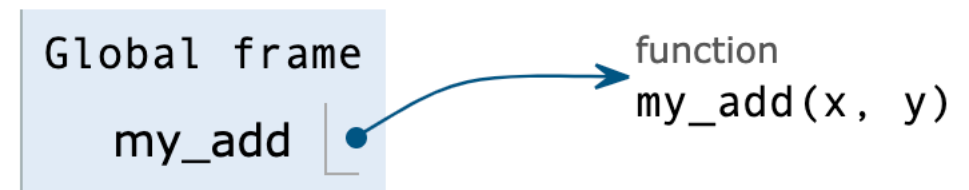
函数是对象

- 函数也是对象，所以可以
 - 把函数分配给变量
 - 把函数存储在数据结构（比如列表、字典）中
 - 作为参数传递给其他函数
 - 作为其他函数的返回值

把函数分配给变量

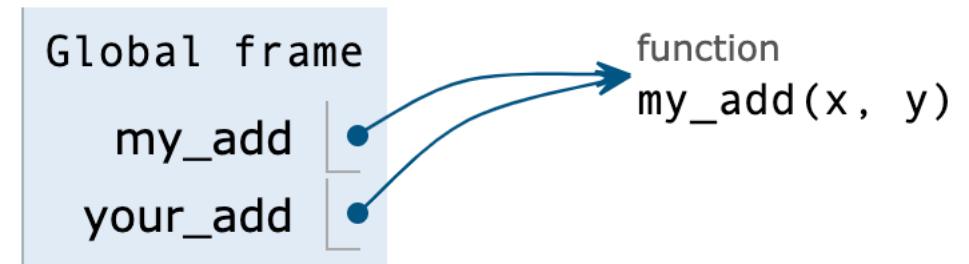
- def语句创建一个函数对象并将其赋值给一个变量
- 赋值语句可以将多个变量绑定到同一个函数

```
>>> def my_add(x, y):  
      return x + y
```



```
>>> my_add(3, 5)  
8
```

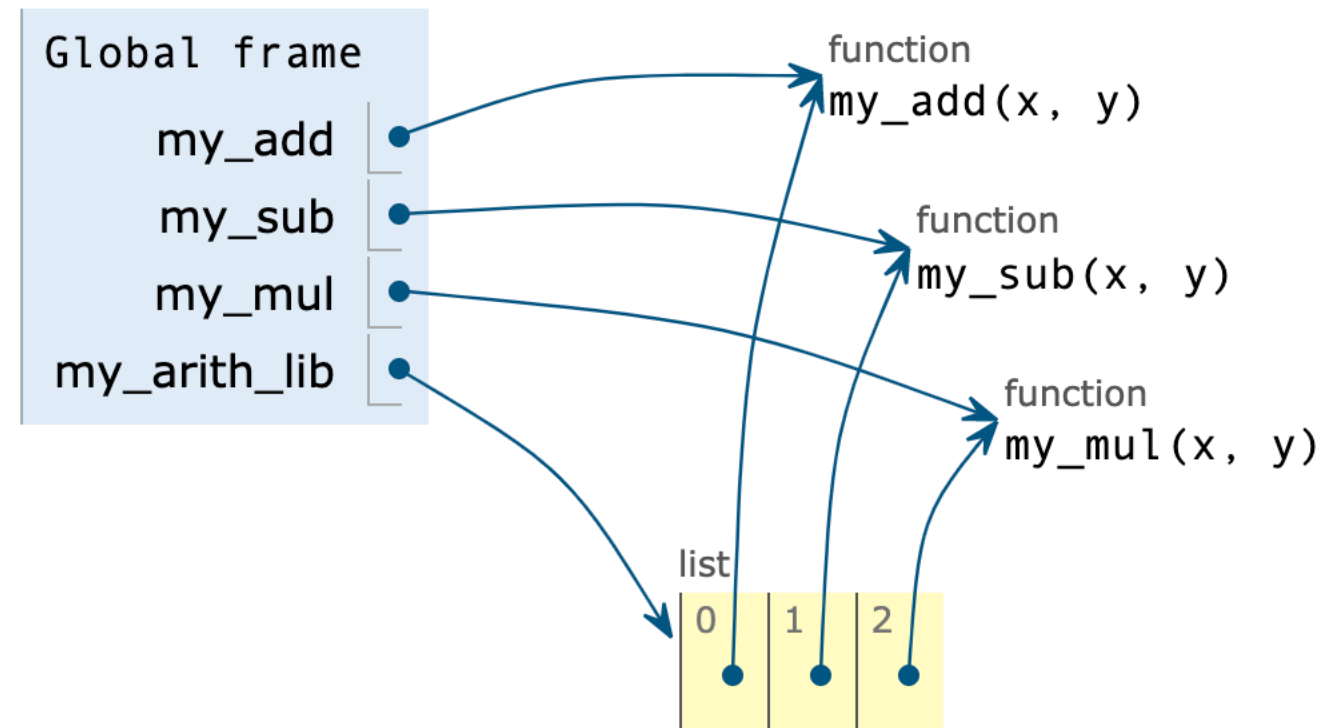
```
>>> your_add = my_add  
>>> your_add(1, 10)  
11
```



函数存储在数据结构中

- 列表是一组任意类型的对象，所以函数也能存放其中

```
>>> def my_sub(x, y):  
    return x - y  
  
>>> def my_mul(x, y):  
    return x * y  
  
>>> my_arith_lib = [my_add,  
                   my_sub,  
                   my_mul,  
                   ]  
  
>>> for f in my_arith_lib:  
    print(f(3, 5), end = ' ')
```



函数作为其他函数的参数

- 不同的参数（传递进来的函数）导致函数具有不同的行为

```
>>> def calculate(f, x, y):  
        return f(x, y)
```

```
>>> calculate(my_add, 3, 5)
```

```
8
```

```
>>> calculate(my_mul, 3, 5)
```

```
15
```

- 接受其他函数作为参数的函数称为高阶函数
- `map(function, iterable, ...)`：返回一个可迭代对象，其中的元素按照下面的规则生成：依次将`iterable`中的元素作为参数来调用函数`function`得到的值的序列

高阶函数map

- `map(function, iterable, ...)` : 返回一个可迭代对象, 其中的元素按照下面的规则生成: 依次将`iterable`中的元素作为参数来调用函数`function`得到的值的序列

```
>>> A = [-2, -1, 0, 1, 2]
```

```
>>> B = [1, 2, 3, 4, 5]
```

```
>>>
```

```
>>> list(map(abs, A)) # 内置函数abs接受一个参数  
[2, 1, 0, 1, 2]
```

```
>>>
```

```
>>> list(map(my_add, A, B)) # my_add接受两个参数  
[-1, 1, 3, 5, 7]
```

函数可以嵌套

- 在函数中可以通过def语句定义另一个函数

```
>>> def maker(n): # 外层函数
        def action(x): # 内层函数
            return x ** n
        return action # 返回函数action, 该函数记住了n的值
```

```
>>> f = maker(2) #调用函数maker, 创建函数action, n的值为2
```

```
>>>
```

```
>>> f(3) # 调用函数action, 参数x为3, n的值为2
```

```
9
```

```
>>> f(4) # 调用函数action, 参数x为4, n的值为2
```

```
16
```

```
>>> g = maker(3) # 创建函数action的另一个版本, n的值为3
```

```
>>>
```

```
>>> g(3) # 调用函数action的另一个版本, n的值为3
```

```
27
```

递归

- 直接或者间接调用自己的函数
- 计算正整数 n 的阶乘： $n! = n \times (n-1) \times (n-2) \cdots \times 2 \times 1$
- 如果函数 $f(n)$ 返回 n 的阶乘，那么 $f(n) = n \times f(n-1)$
- 递归的两种情况
 - 递归情况：调用自己解决一个类似但规模较小的问题
 - 要计算 $f(10)$ ，只需计算 $f(9)$ ，然后返回 $10 \times f(9)$
 - 基本情况：直接可以得到解，无需再次调用自己
 - 要计算 $f(1)$ ，根据定义知道1的阶乘是1

递归

```
>>> def my_fac(n):  
    if n == 1:  
        return 1  
    else:  
        return n * my_fac(n-1)
```

```
>>> my_fac(5)
```

```
120
```

```
>>> my_fac(20)
```

```
2432902008176640000
```

```
>>> import math
```

```
>>> math.factorial(20)
```

```
2432902008176640000
```



如果调用my_fac函数的参数是0或者-1，情况会如何？

→ line that has just executed

→ next line to execute

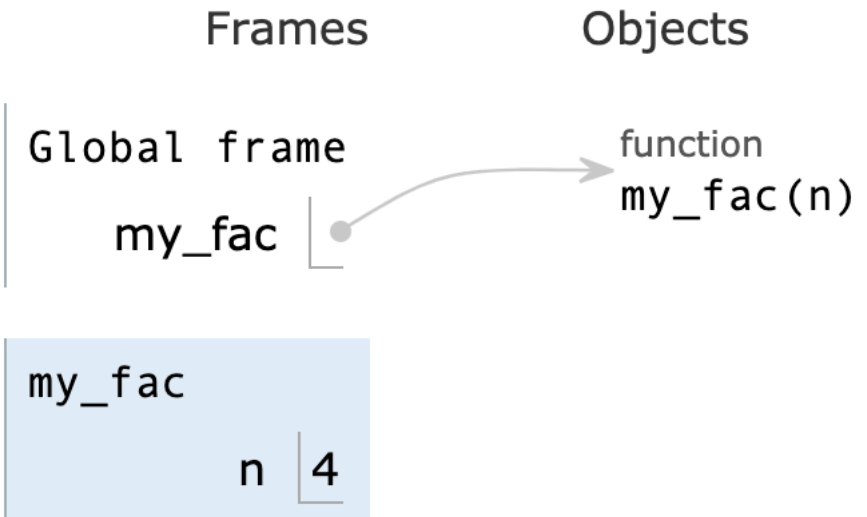
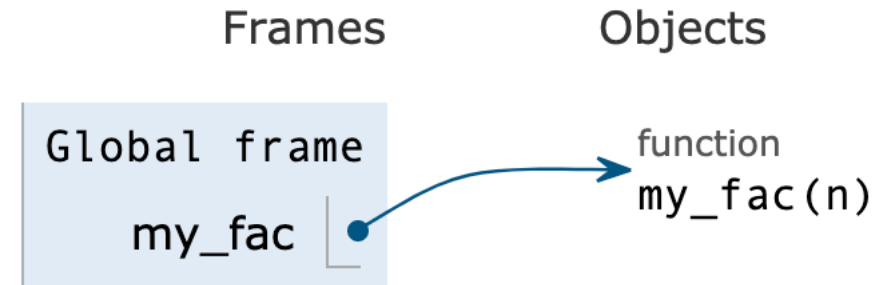
```
→ 1 def my_fac(n):  
  2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6
```

(1) → 7 res = my_fac(4)

```
→ 1 def my_fac(n):  
  2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6
```

(2) → 7 res = my_fac(4)

```
→ 1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6  
(3) 7 res = my_fac(4)
```



```
1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
(4) 7 res = my_fac(4)
```

→ line that has just executed

→ next line to execute

(5)

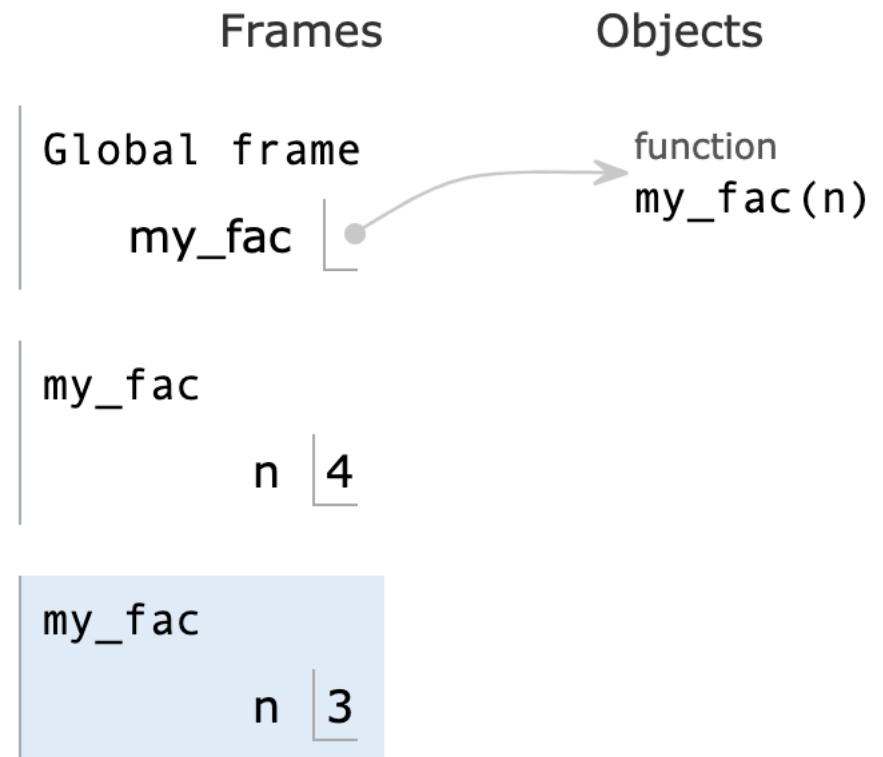
```
→ 1 def my_fac(n):  
  2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(6)

```
→ 1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(7)

```
  1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```



→ line that has just executed

→ next line to execute

(8)

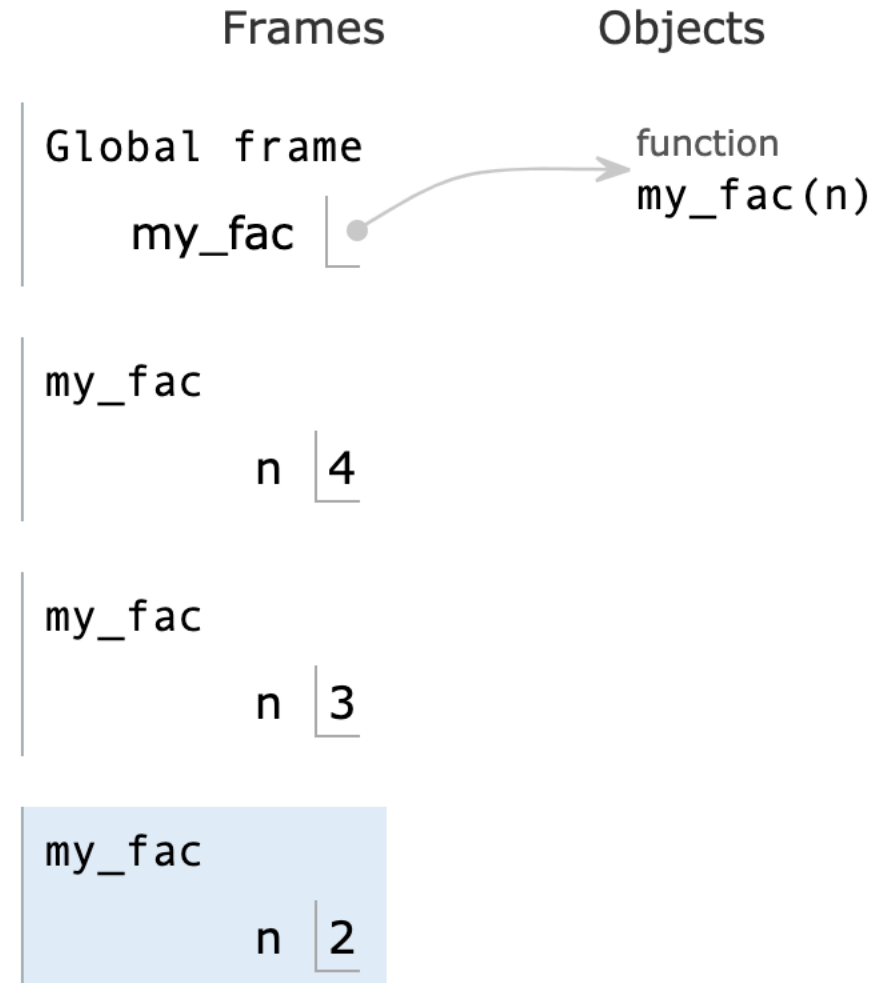
```
→ 1 def my_fac(n):  
  2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(9)

```
→ 1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(10)

```
  1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```



→ line that has just executed

→ next line to execute

(11)

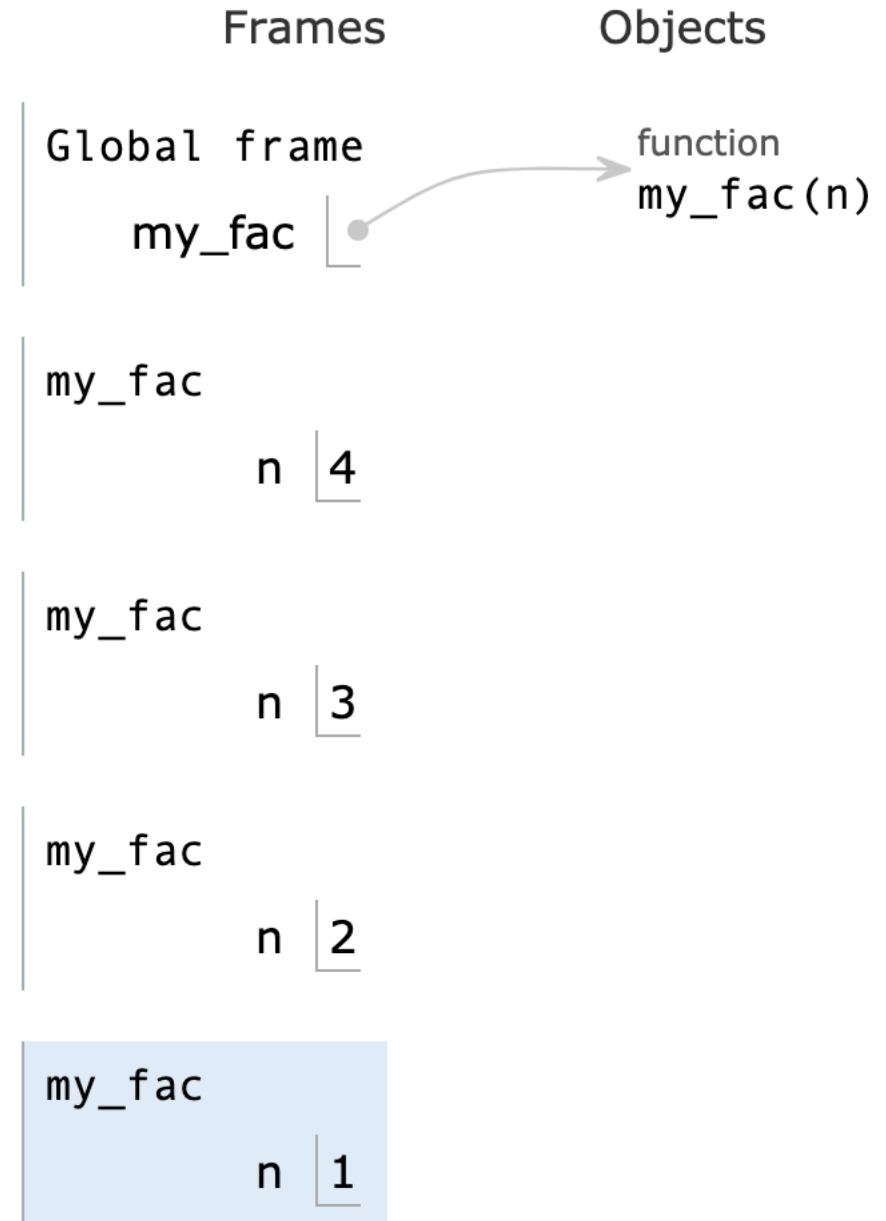
```
→ 1 def my_fac(n):  
  2     if n == 1:  
  3         return 1  
  4     else:  
→ 5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(12)

```
→ 1 def my_fac(n):  
→ 2     if n == 1:  
  3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

(13)

```
  1 def my_fac(n):  
→ 2     if n == 1:  
→ 3         return 1  
  4     else:  
  5         return n * my_fac(n-1)  
  6  
  7 res = my_fac(4)
```

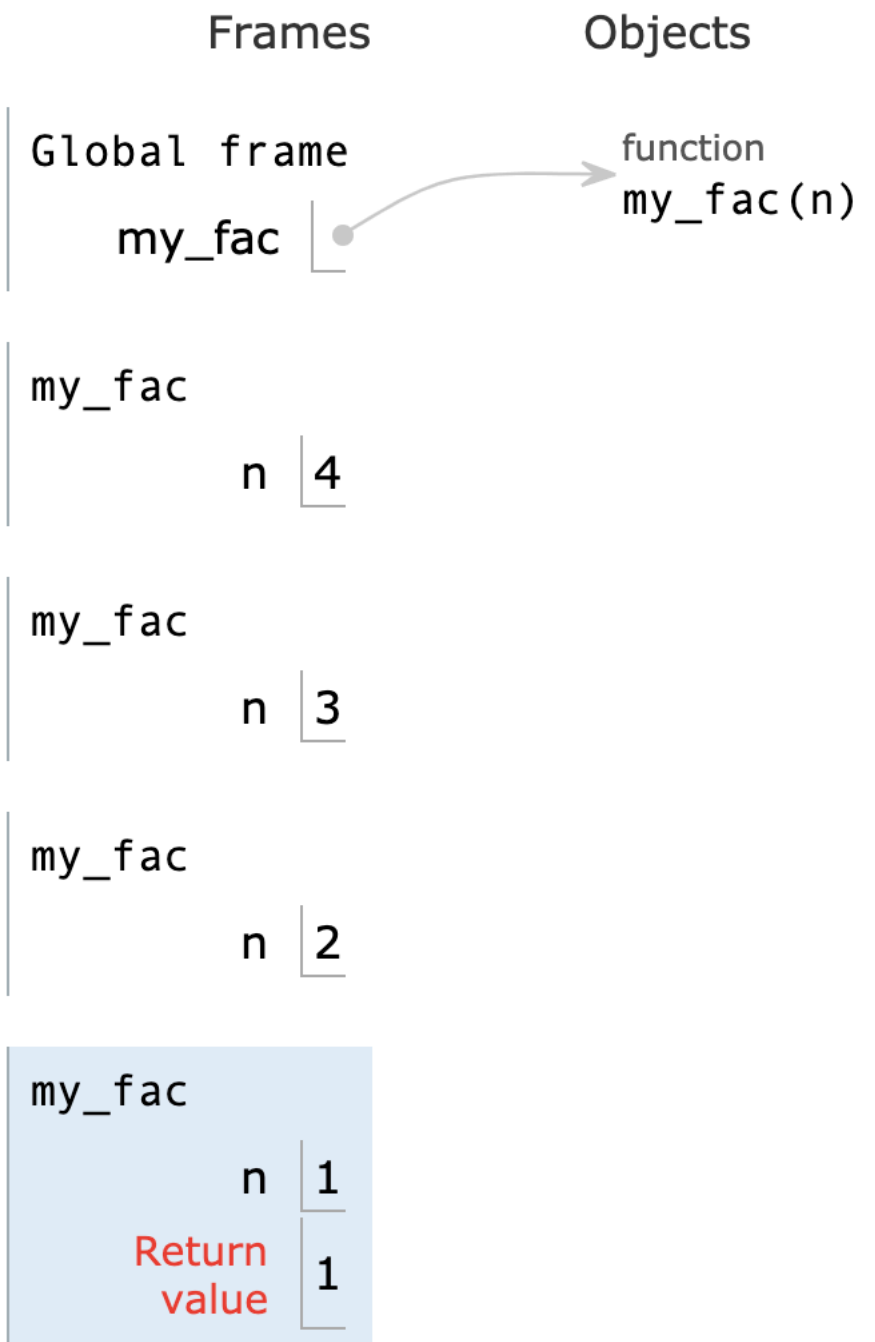


(14)

```

1 def my_fac(n):
2     if n == 1:
3         return 1
4     else:
5         return n * my_fac(n-1)
6
7 res = my_fac(4)

```

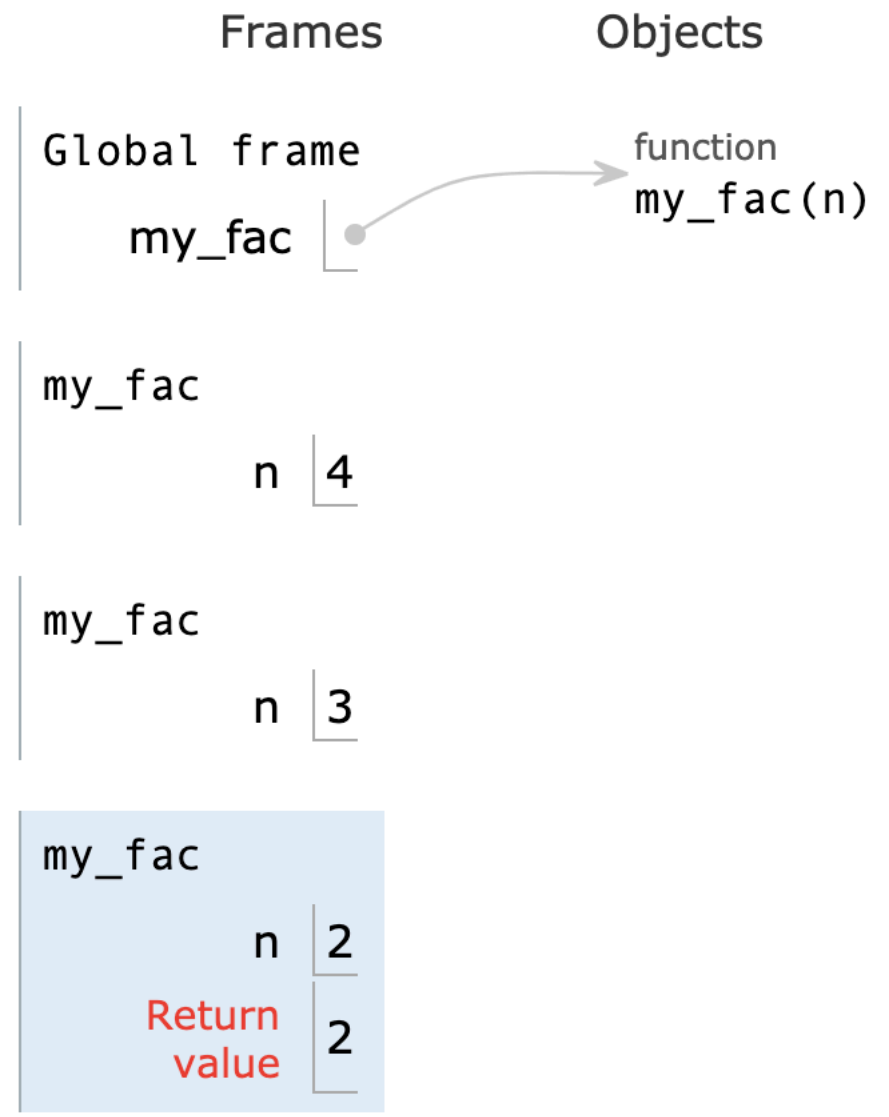


(15)

```

1 def my_fac(n):
2     if n == 1:
3         return 1
4     else:
5         return n * my_fac(n-1)
6
7 res = my_fac(4)

```



→ line that has just executed
→ next line to execute

→ line that has just executed

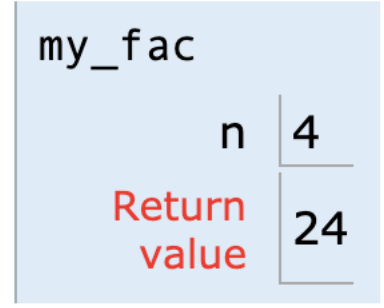
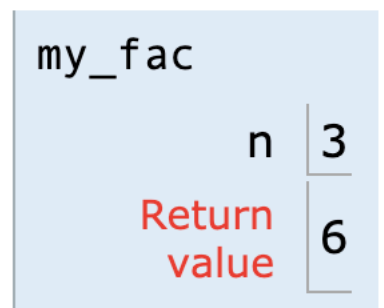
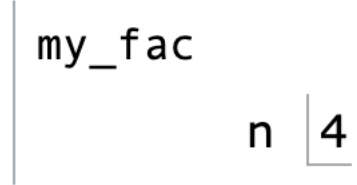
→ next line to execute

(16)

```

1 def my_fac(n):
2     if n == 1:
3         return 1
4     else:
5         return n * my_fac(n-1)
6
7 res = my_fac(4)

```

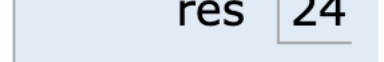


(17)

```

1 def my_fac(n):
2     if n == 1:
3         return 1
4     else:
5         return n * my_fac(n-1)
6
7 res = my_fac(4)

```



(18)

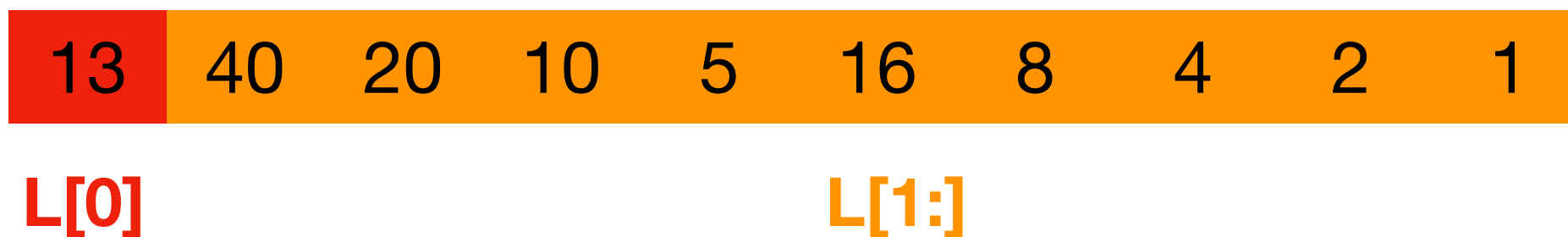
```

1 def my_fac(n):
2     if n == 1:
3         return 1
4     else:
5         return n * my_fac(n-1)
6
7 res = my_fac(4)

```

列表求和

- 编写一个函数my_sum, 接受一个列表L, 求所有元素之和
- 要求: 不使用内置sum函数, 使用递归
- 基本情况 - 列表为空, 返回0
- 递归情况 - 列表不为空, 返回L[0]+my_sum(L[1:])



用递归求和

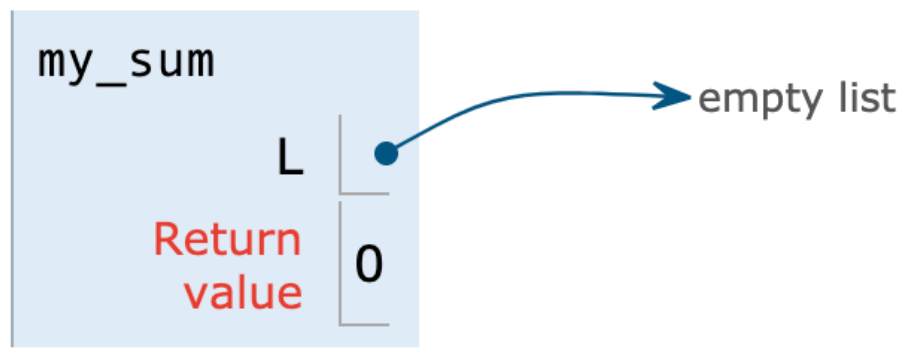
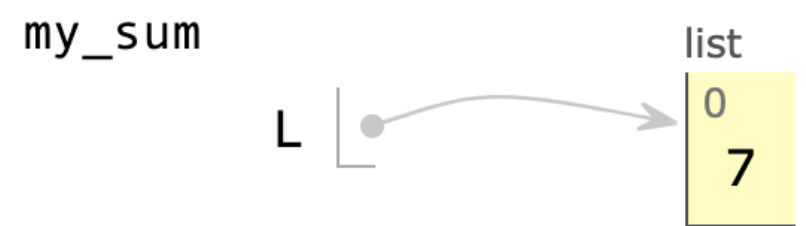
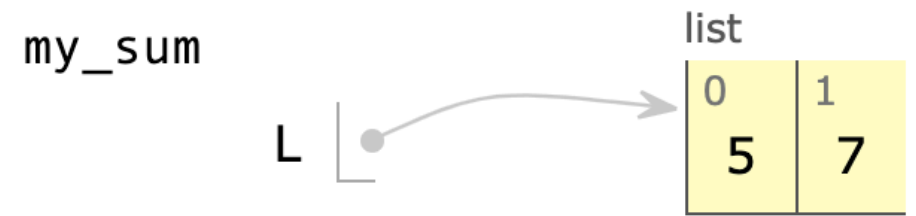
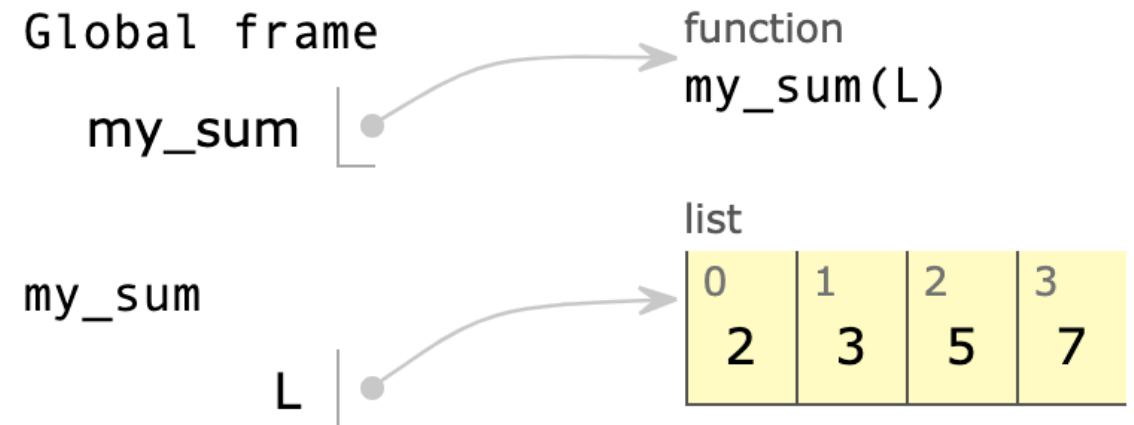
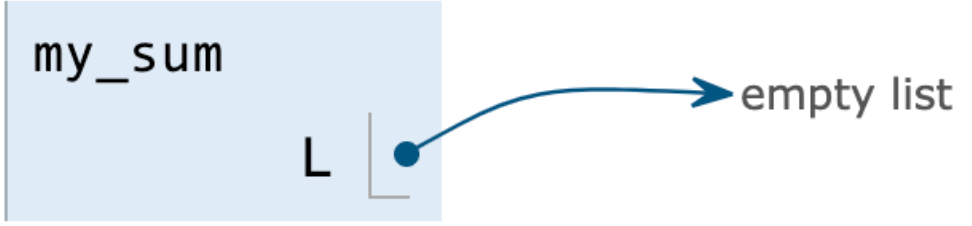
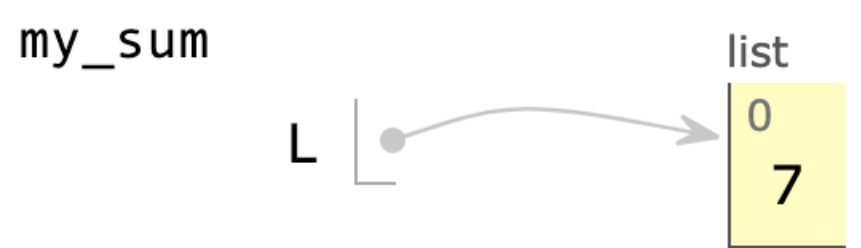
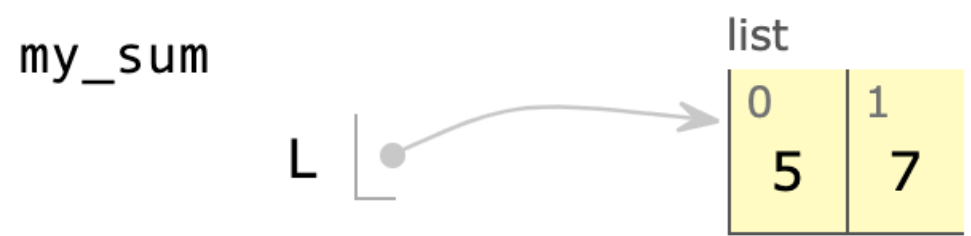
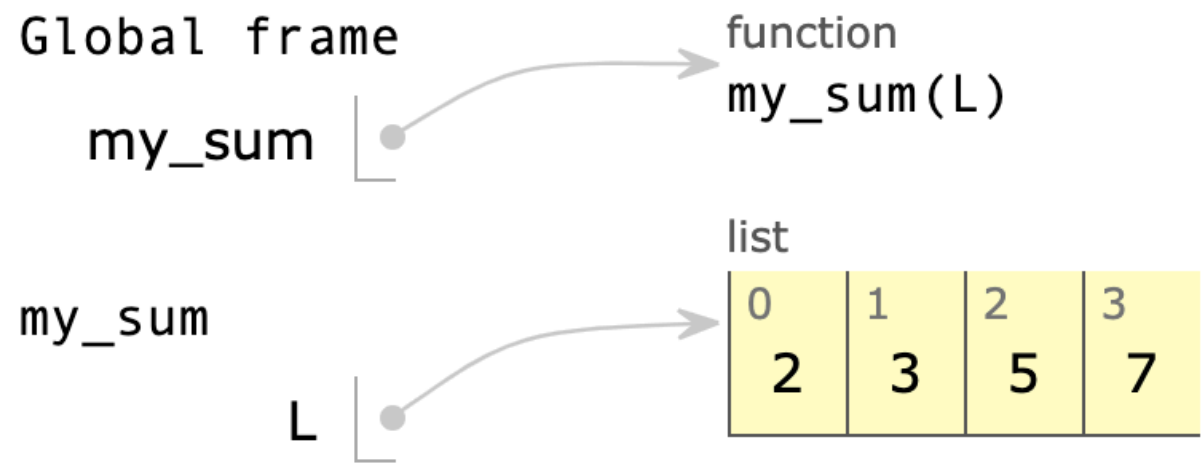
- 编写一个函数my_sum, 接受一个列表L, 用递归求和
- 基本情况 - 列表为空, 返回0
- 递归情况 - 列表不为空, 返回L[0]+my_sum(L[1:])

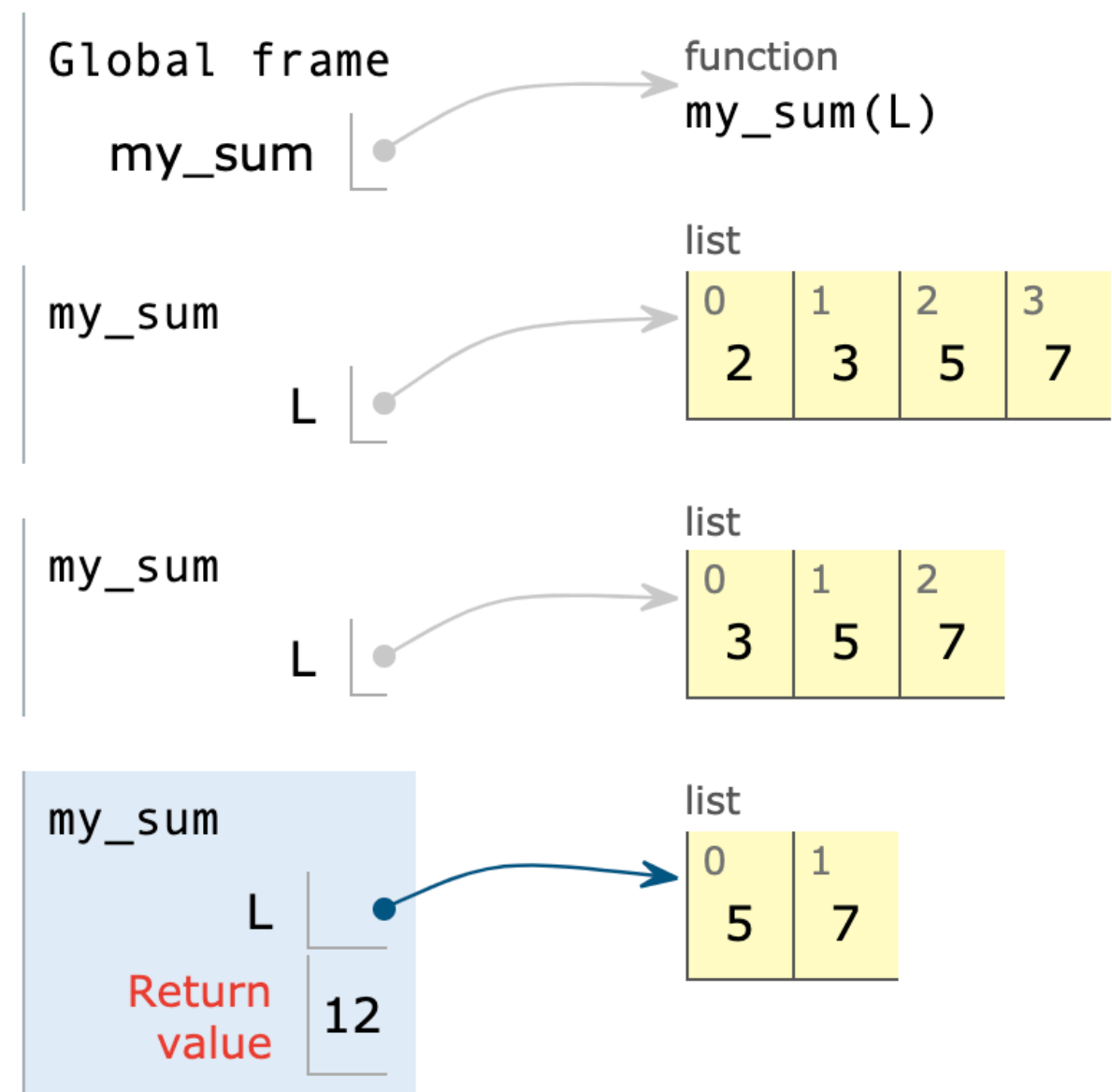
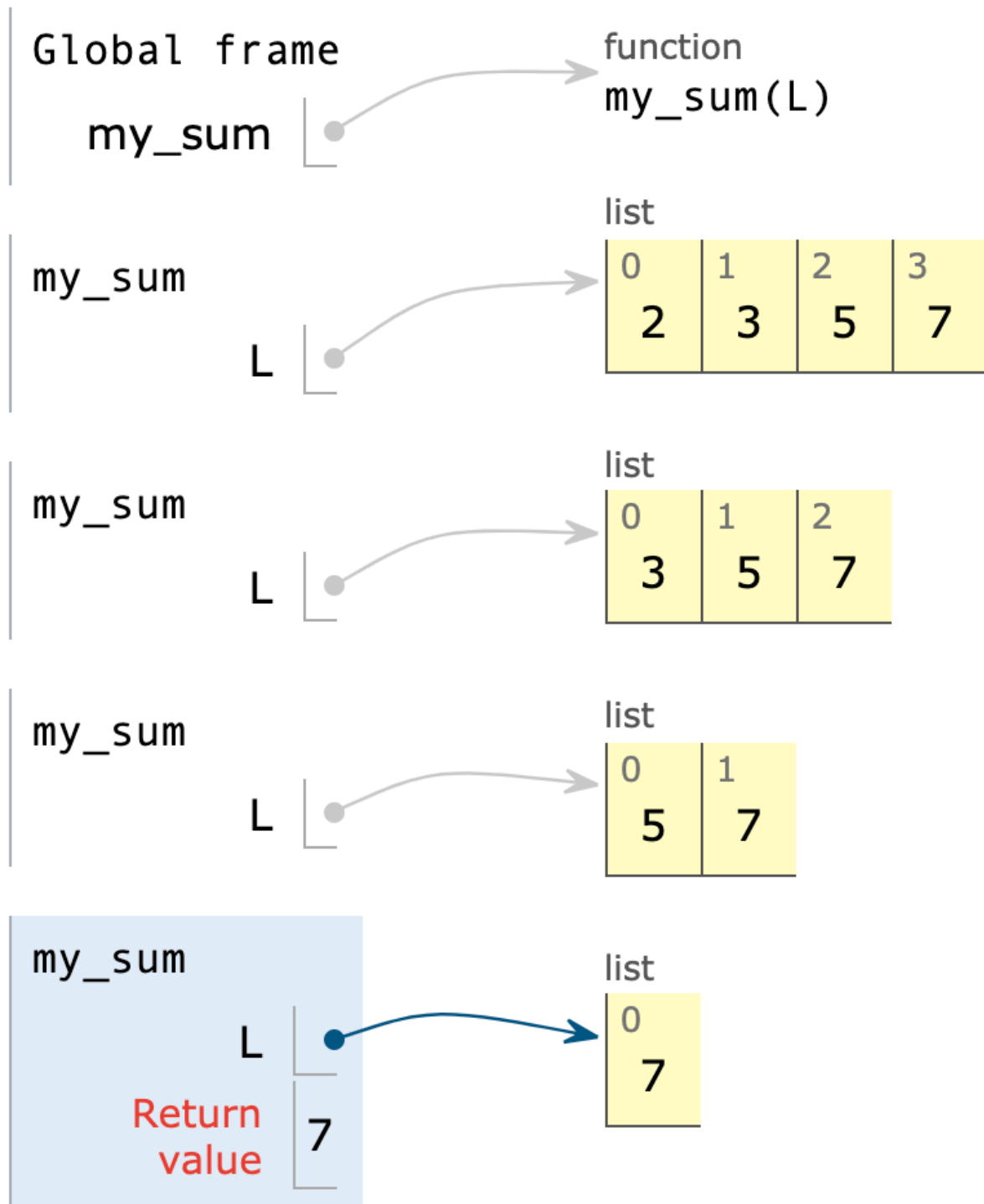
```
1 def my_sum(L):  
2     if not L:  
3         return 0  
4     else:  
5         return L[0] + my_sum(L[1:])
```

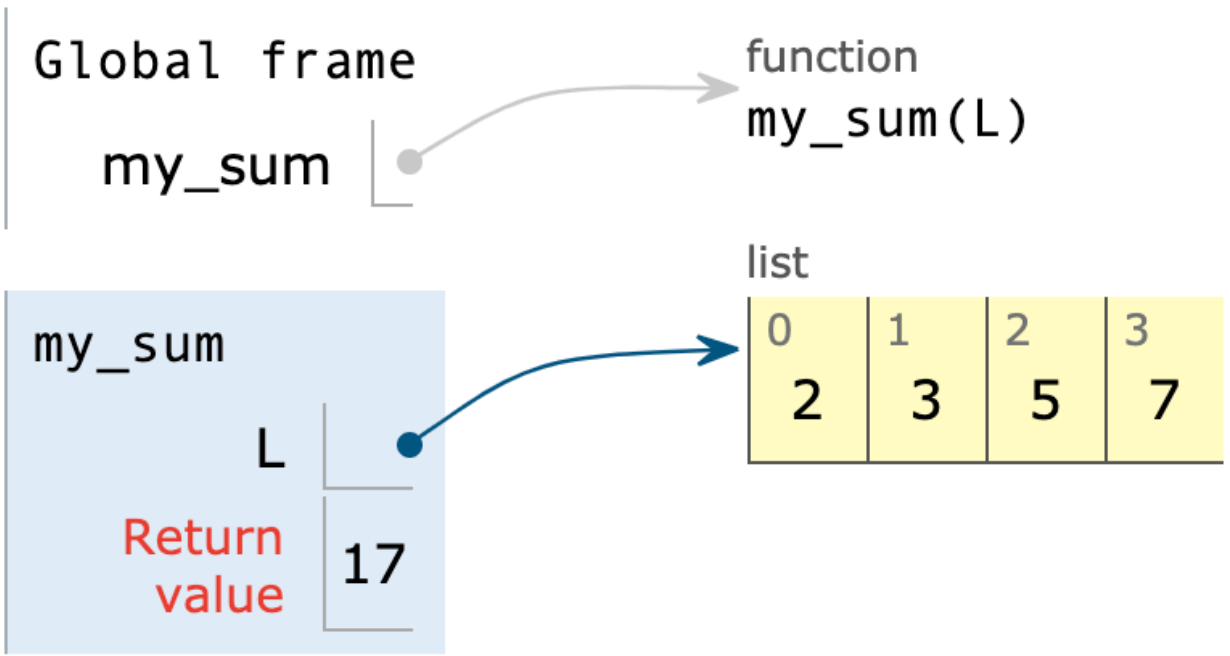
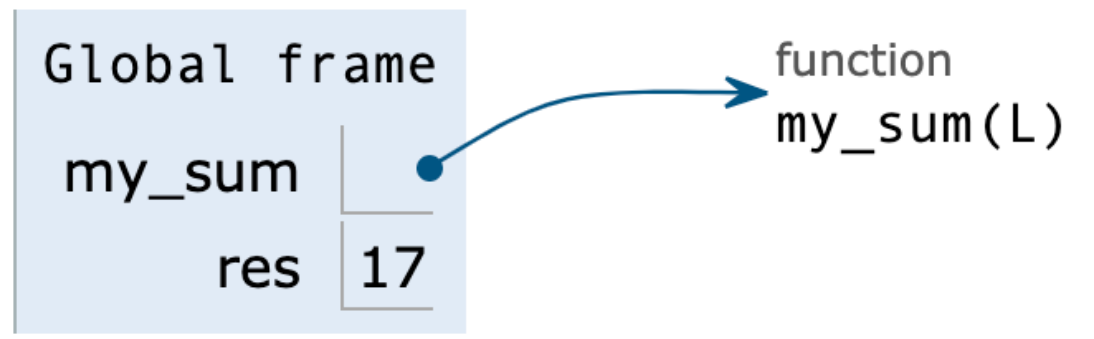
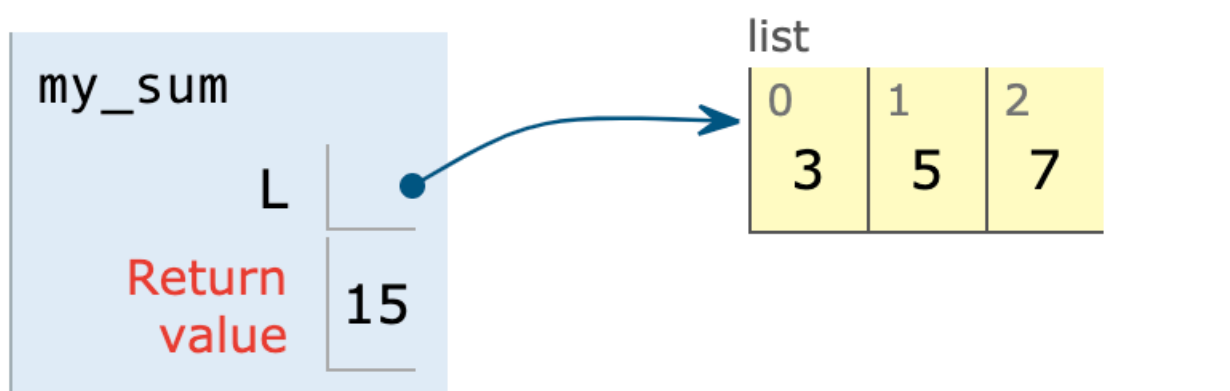
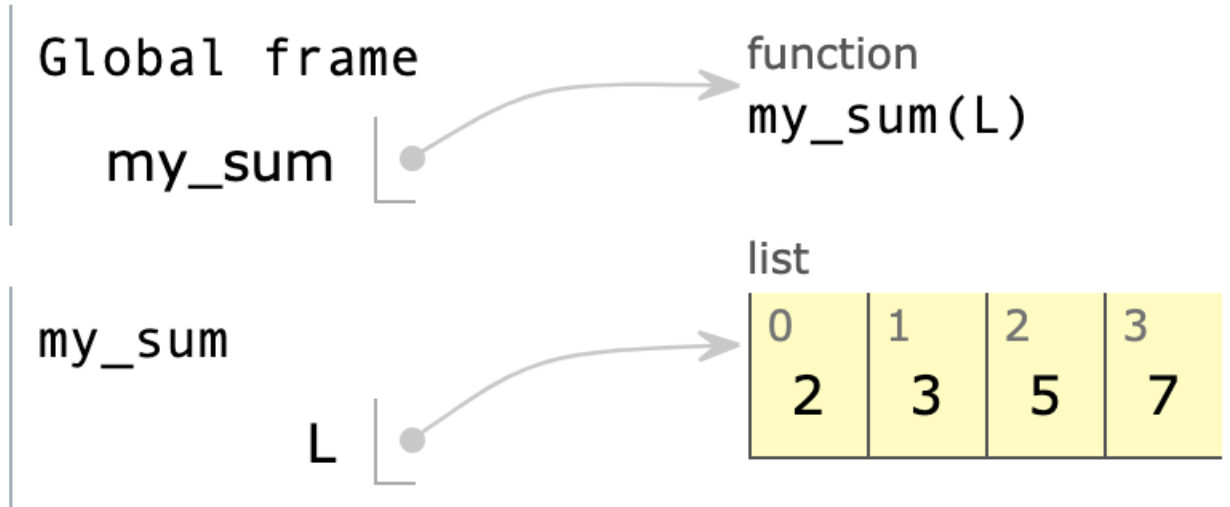
```
>>> res = my_sum([2, 3, 5, 7])
```

```
>>> res
```

```
17
```







嵌套列表求和

- 内置sum函数可以对一个列表求和
- 注意：如果列表的元素不能相加，则会报错

```
>>> L = [2, 3, 5, 7]
```

```
>>> sum(L)
```

```
17
```

```
>>> A = [1, [2, [3, 4], 5], 6, [7, 8]]
```

```
>>> sum(A)
```

```
Traceback (most recent call last):
```

```
  File "<pysHELL#69>", line 1, in <module>
```

```
    sum(A)
```

```
TypeError: unsupported operand type(s) for +:  
'int' and 'list'
```

嵌套列表求和

- 编写一个函数`my_sum_v1`，接受一个嵌套列表（嵌套深度随意），求该列表中所有数值的和
- 基本情况 - 当前元素是一个数值，直接返回
- 递归情况 - 当前元素是一个列表
 - 对于列表中的每一个元素，使用`my_sum_v1`求和
 - 遍历完成之后，返回总的和
- 如何判断一个元素是否是列表？
 - 使用内置函数`isinstance(obj, classinfo)`进行判断

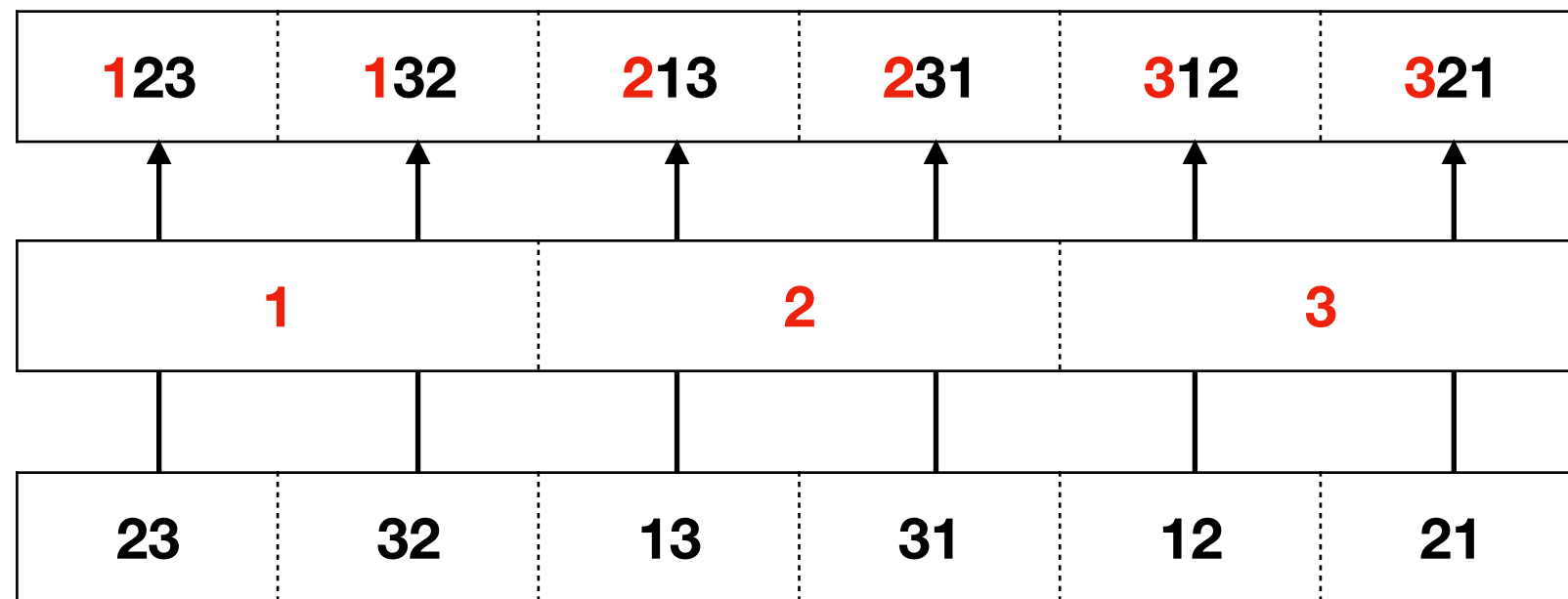
嵌套列表求和

- 基本情况 - 当前元素是一个数值，直接返回
- 递归情况 - 当前元素是一个列表
 - 对于列表中的每一个元素，使用my_sum_v1求和
 - 遍历完成之后，返回总的和

```
1 def my_sum_v1(L):
2     if not isinstance(L, list):
3         return L
4     else:
5         total = 0
6         for x in L:
7             total = total + my_sum_v1(x)
8         return total
```

全排列

- 给定n个元素，生成它们的全排列。比如元素1,2,3的全排列共有6种情况：123、132、213、231、312、321
- 2个元素a和b的全排列只有两种情况：ab和ba
- 3个元素的全排列如何由2个元素的全排列得到？



全排列

- 给定 n 个元素，生成它们的全排列。比如元素1,2,3的全排列共有6种情况：123、132、213、231、312、321
- 假设函数permute(L)可以生成列表L中所有元素的全排列
- n 个元素的全排列如何生成？
 - 任一元素 + 剩余 $n-1$ 个元素的全排列
 - $L[0] + \text{permute}(L[1:])$
 - $L[i] + \text{permute}(L[:i] + L[i+1:])$
 - $L[n-1] + \text{permute}(L[:n-1])$

全排列

```
>>> def permute(L):
    if not L:
        return [L] # 应该返回 [[]]
    else:
        res = []
        for i in range(len(L)):
            rest = L[:i] + L[i+1:]
            # 如果rest为空, permute函数应该返回 [[]]
            for x in permute(rest):
                res.append(L[i:i+1] + x)
        return res
```

```
>>> permute([1,2,3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
>>> len(permute([1,2,3,4,5]))
120
```