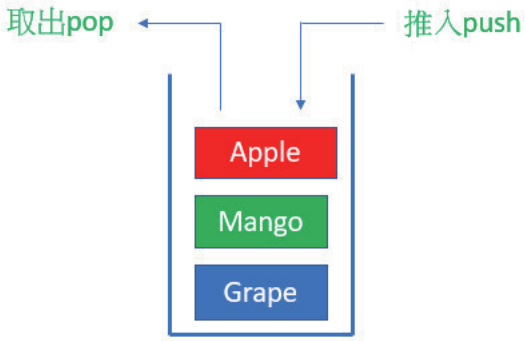
第 5 章

栈

栈 (stack) 也是一个线性的数据结构，特色是由下往上堆放数据，如下所示：

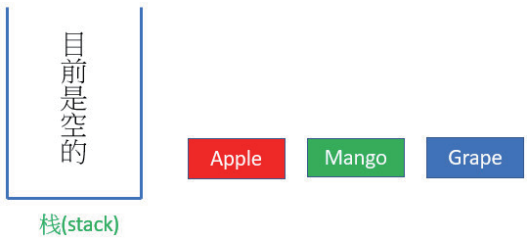


将数据插入栈的动作称推入 (push)，动作是由下往上堆放。将数据从栈中读取的动作称取出 (pop)，动作是由上往下读取，数据经读取后同时从栈中移除。由于每一个数据皆从同一端进入与离开栈，整个过程有先进后出 (first in last out) 的特征。

每一个程序语言的递归式调用 (recursive call)，其设计原理就是栈，未来笔者还会做更多的解析。

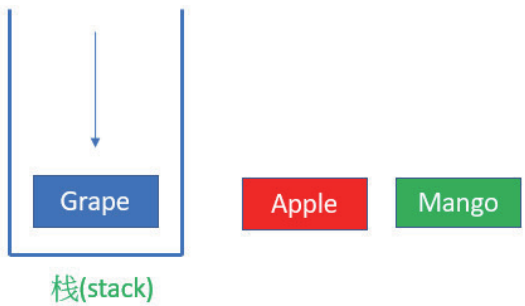
5-1 数据推入 push

假设我们依序要推入 Grape、Mango、Apple，整个步骤说明如下：



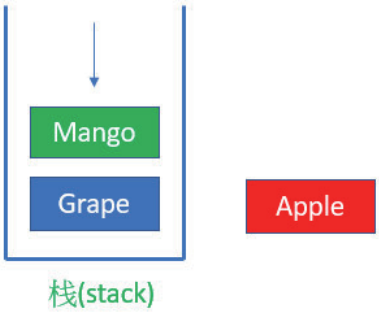
❑ 步骤 1：

将 Grape 推入栈。



❑ 步骤 2：

将 Mango 推入栈。



❑ 步骤 3：

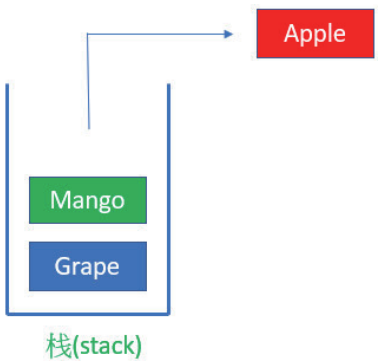
将 Apple 推入栈。

5-2 数据取出 pop

读取数据后将此数据从栈中移除，我们也可以称此过程为读取数据，下列是依序读取栈内数据的步骤说明：

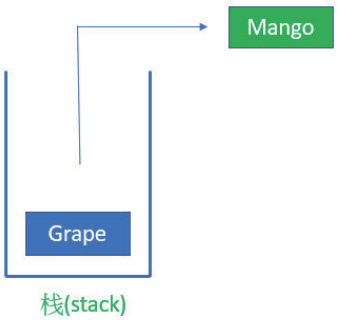
❑ 步骤 1：

取出数据，可以得到 Apple，同时 Apple 从栈中被移除。



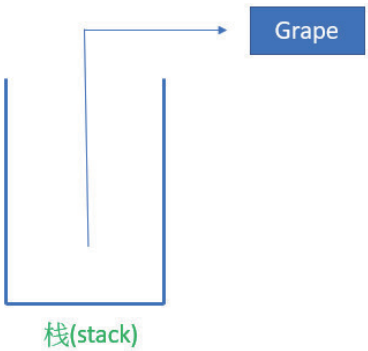
❑ 步骤 2：

取出数据，可以得到 Mango，同时 Mango 从栈中被移除。



❑ 步骤 3：

取出数据，可以得到 Grape，同时 Grape 从栈中被移除。



这种数据结构的特点是必须先读取最后进入的数据，无法读取中间数据，未来我们还会用实例讲解这类数据结构的应用。

5-3 Python 中栈的应用

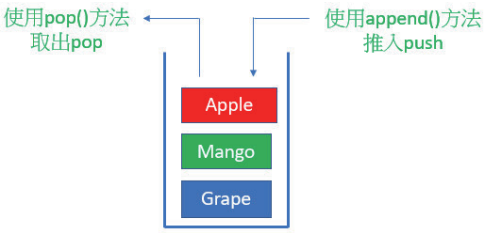
Python 的列表 (list) 结构可以让我们很方便地实现前两节的栈操作，在这一节笔者将讲解使用 Python 内建列表直接模拟栈操作，以及使用列表功能重新诠释栈操作，同时我们也可以增加一些功能操作，下列将一一讲解。

5-3-1 使用列表 (list) 模拟栈操作

在 Python 程序语言中关于列表 (list) 有两个很重要的内建方法：

append( )：在列表末端加入数据，读者可以想成是栈的 push 方法。

pop( )：读取列表末端的数据同时删除该数据，读者可以想成是栈的 pop 方法。



程序实例 ch5_1.py：使用 Python 的 append( ) 模拟栈的 push，使用 Python 的 pop( ) 模拟栈的 pop。

```
# ch5_1.py
fruits = []
fruits.append('Grape')
fruits.append('Mango')
fruits.append('Apple')
print('打印 fruits = ', fruits)
print('pop操作 : ', fruits.pop())
print('pop操作 : ', fruits.pop())
print('pop操作 : ', fruits.pop())
```

执行结果

```
====================== RESTART: D:\Algorithm\ch5\ch5_1.py ======================
打印 fruits =  ['Grape', 'Mango', 'Apple']
pop操作 :  Apple
pop操作 :  Mango
pop操作 :  Grape
```

5-3-2　自行建立 stack 类别执行相关操作

程序实例 ch5_2.py：将 Grape、Mango、Apple 分别推入栈，然后输出有多少种水果在栈内。

```
# ch5_2.py
class Stack():
    def __init__(self):
        self.my_stack = []

    def my_push(self, data):
        self.my_stack.append(data)

    def my_pop(self):
        return self.my_stack.pop()

    def size(self):
        return len(self.my_stack)

stack = Stack()
fruits = ['Grape', 'Mango', 'Apple']
for fruit in fruits:
    stack.my_push(fruit)
    print('将 %s 水果推入栈' % fruit)

print('栈有 %d 种水果' % stack.size())
```

执行结果

```
====================== RESTART: D:\Algorithm\ch5\ch5_2.py ======================
将Grape 水果推入栈
将Mango 水果推入栈
将Apple 水果推入栈
栈有 3 种水果
```

程序实例 ch5_3.py：扩充设计 ch5_2.py，将数据推入栈输出数量后，再将数据取出。在这个程序设计中，为了确认所有数据是否都已经取出，可以在 Stack 类别内增加 isEmpty( ) 方法。

```
1  # ch5_3.py
2  class Stack():
3      def __init__(self):
4          self.my_stack = []
5
6      def my_push(self, data):
7          self.my_stack.append(data)
8
9      def my_pop(self):
10         return self.my_stack.pop()
11
12     def size(self):
13         return len(self.my_stack)
14
15     def isEmpty(self):
16         return self.my_stack == []
17
18 stack = Stack()
19 fruits = ['Grape', 'Mango', 'Apple']
20 for fruit in fruits:
21     stack.my_push(fruit)
22     print('将 %s 水果推入栈' % fruit)
23
24 print('栈有 %d 种水果' % stack.size())
25 while not stack.isEmpty():
26     print(stack.my_pop())
```

执行结果

```
====================== RESTART: D:\Algorithm\ch5\ch5_3.py ======================
将Grape 水果推入栈
将Mango 水果推入栈
将Apple 水果推入栈
栈有 3 种水果
Apple
Mango
Grape
```

5-4 函数调用与栈运作

计算机语言在执行函数调用时，内部其实是使用栈在运作，下列将以实例做说明。

程序实例 ch5_4.py：由函数调用了解程序语言的运作。

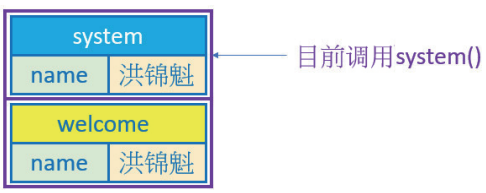
```
1  # ch5_4.py
2  def bye():
3      print("下回见!")
4
5  def system(name):
6      print("%s 欢迎进入校友会系统" % name)
7
8  def welcome(name):
9      print("%s 欢迎进入明志科技大学系统" % name)
10     system(name)
11     print("使用明志科技大学系统很棒")
12     bye()
13
14 welcome("洪锦魁")
```

执行结果

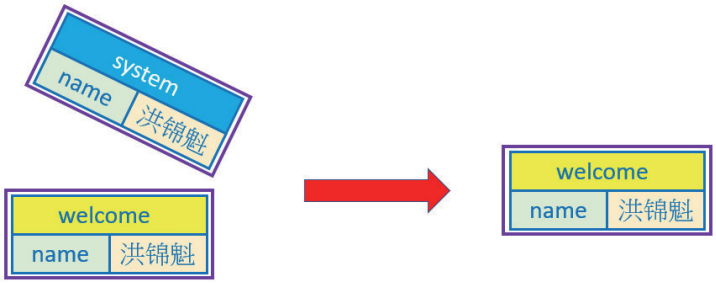
```
====================== RESTART: D:\Algorithm\ch5\ch5_4.py ======================
洪锦魁 欢迎进入明志科技大学系统
洪锦魁 欢迎进入校友会系统
使用明志科技大学系统很棒
下回见!
```

上述是一个简单的调用函数程序，接下来我们看这个程序如何应用栈运作。程序第 14 行调用 welcome( ) 时，计算机内部会以栈方式配置一个内存空间。

当有函数调用时，计算机会将调用的函数名称与所有相关的变量存储在内存内，然后进入 welcome( ) 函数。当执行第 9 行时会输出“洪锦魁 欢迎进入明志科技大学系统”。当执行第 10 行时调用 system( )，计算机内部会以栈方式配置一个内存空间，同时堆放在前一次调用的 welcome( ) 内存上方。

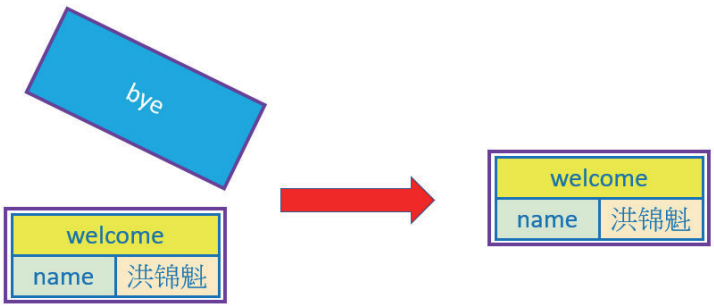


程序接着执行第 6 行，输出“洪锦魁 欢迎进入校友会系统”，然后 system( ) 函数执行结束，此时程序返回 welcome( ) 函数，同时将上方的内存移除，回到 welcome( ) 函数。



上图有一个很重要的概念是，welcome( ) 函数执行一半时，工作先暂停但是内存数据仍然保留，先去执行另一个函数 system( )。当 system( ) 工作结束时，可以回到 welcome( ) 函数先前暂停的位置继续往下执行。接着执行第 11 行输出“使用明志科技大学系统很棒”。然后执行第 12 行调用 bye( ) 函数，这个调用没有传递变量，栈内存如下所示：

系统会将 bye( ) 函数新增在栈上方，然后执行第 3 行输出“下回见！”，接着 bye( ) 函数执行结束。此时程序返回 welcome( ) 函数，同时将上方的内存移除，回到 welcome( ) 函数。

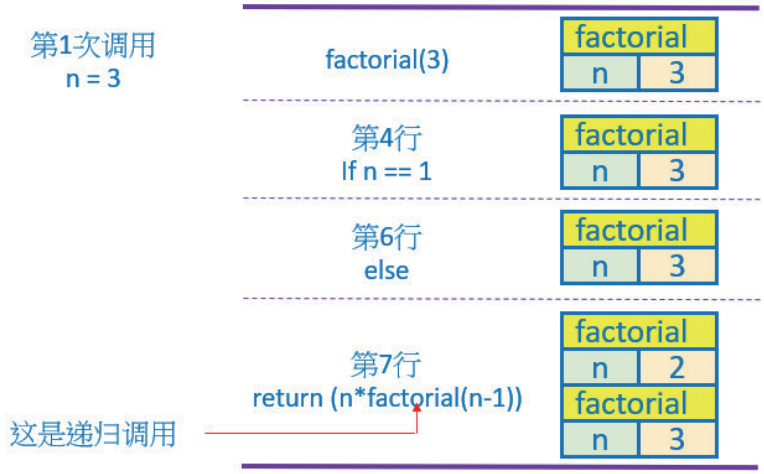


从调用 bye( ) 到返回 welcome( ) 函数后，由于 welcome( ) 函数也执行结束，所以整个程序就算执行结束了。

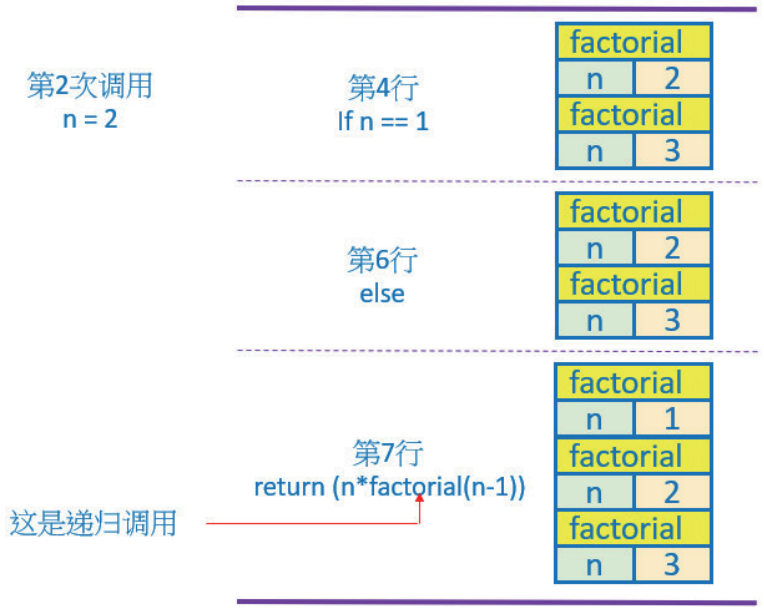
5-5 递归调用与栈运作

本书程序 ch1_1.py 使用递归调用计算阶乘，笔者输入阶乘数 n=3，然后程序第 10 行调用 factorial(n) 函数，此时栈内存内容如下：

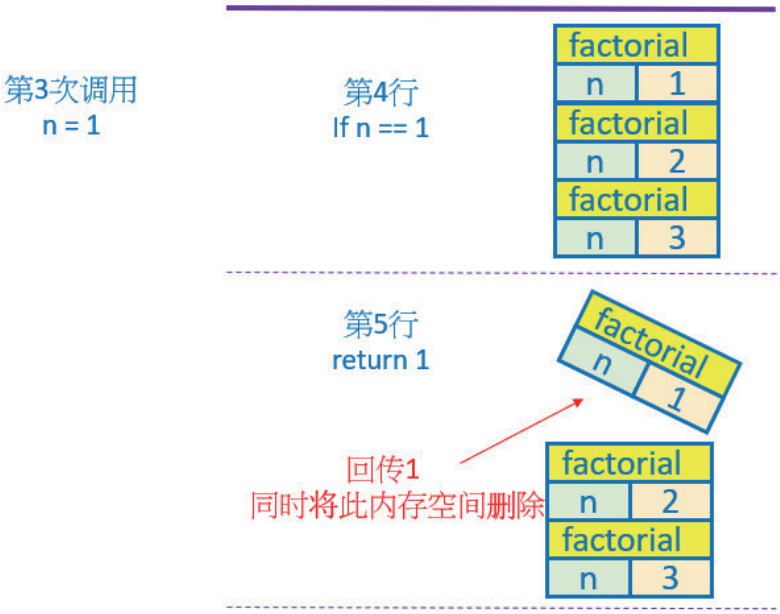
接着进入 factorial(3) 函数，此时程序代码与栈内存内容如下：



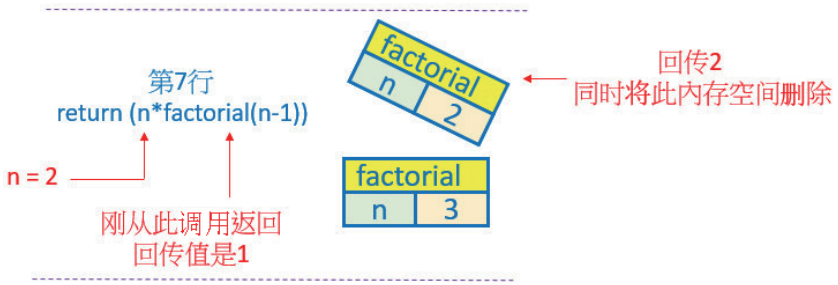
下列是第 2 次调用 factorial(2) 函数，此时程序代码与栈内存内容如下：



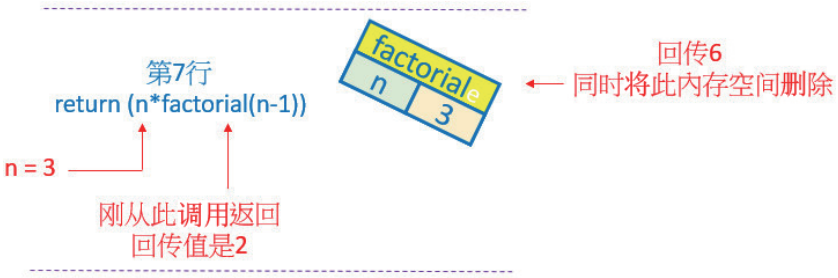
下列是第 3 次调用 factorial(1) 函数，此时程序代码与栈内存内容如下：



下列是返回的操作：



下列是再一次返回的操作：



所以程序实例 ch1_1.py 可以得到 6 的结果。在算法中有关递归调用与栈的应用仍有许多，本书未来还会有实例说明。

程序实例 ch5_5.py：这是 ch1_1.py 的改良，主要是在 factorial( ) 函数内增加注释，读者可以从此函数看到递归调用的计算过程。

```
# ch5_5.py
def factorial(n):
    global fact
    """ 计算n的阶乘，n 必须是正整数 """
    if n == 1:
        print("factorial(%d)调用前 %d! = %d" % (n, n, fact))
        print("到达递归条件终止 n = 1")
        fact = 1
        print("factorial(%d)返回后 %d! = %d" % (n, n, fact))
        return fact
    else:
        print("factorial(%d)调用前 %d! = %d" % (n, n, fact))
        fact = n * factorial(n-1)
        print("factorial(%d)返回后 %d! = %d" % (n, n, fact))
        return fact

fact = 0
N = eval(input("请输入阶乘数 : "))
print(N, " 的阶乘结果是 = ", factorial(N))
```

执行结果

```
====================== RESTART: D:\Algorithm\ch5\ch5_5.py ======================
请输入阶乘数 : 9
factorial(9)调用前 9! = 0
factorial(8)调用前 8! = 0
factorial(7)调用前 7! = 0
factorial(6)调用前 6! = 0
factorial(5)调用前 5! = 0
factorial(4)调用前 4! = 0
factorial(3)调用前 3! = 0
factorial(2)调用前 2! = 0
factorial(1)调用前 1! = 0
到达递归条件终止 n = 1
factorial(1)返回后 1! = 1
factorial(2)返回后 2! = 2
factorial(3)返回后 3! = 6
factorial(4)返回后 4! = 24
factorial(5)返回后 5! = 120
factorial(6)返回后 6! = 720
factorial(7)返回后 7! = 5040
factorial(8)返回后 8! = 40320
factorial(9)返回后 9! = 362880
9  的阶乘结果是 =  362880
```

5-6 习题

1. 请为程序实例 ch5_3.py 的 Stack 类别设计方法 get( )，这个方法可以传回栈顶端的值，同时数据不删除，请执行 3 次，然后再参考 ch5_3.py 将栈的数据取出。

```
====================== RESTART: D:\Algorithm\ex\ex5_1.py ======================
将Grape 水果推入栈
将Mango 水果推入栈
将Apple 水果推入栈
栈有 3 种水果
取出 Apple 水果, 同时不删除
取出 Apple 水果, 同时不删除
取出 Apple 水果, 同时不删除
Apple
Mango
Grape
```

2. 请为程序实例 ch5_3.py 的 Stack 类别设计方法 cls( )，这个方法可以删除所有栈数据。请在将数据推入栈后，先列出栈中数据的数量，然后调用 cls( ) 方法，最后保持原先第 25 ～ 26 行打印栈的设计，程序末端增加打印“程序结束”，这时可以看到打印栈时没有数据显示。

```
====================== RESTART: D:\Algorithm\ex\ex5_2.py ======================
将Grape 水果推入栈
将Mango 水果推入栈
将Apple 水果推入栈
栈有 3 种水果
程序结束
```