函数（二）

- 目标
  - 冒泡排序
  - 递归
  - lambda 表达式
  - 高阶函数

1、交换变量值

需求：有变量`a = 10`和`b = 20`，交换两个变量的值。

- 方法一

借助第三变量存储数据。

```
# 1. 定义中间变量
c = 0

# 2. 将a的数据存储到c
c = a

# 3. 将b的数据20赋值到a，此时a = 20
a = b

# 4. 将之前c的数据10赋值到b，此时b = 10
b = c

print(a)  # 20
print(b)  # 10
```

- 方法二

```
a, b = 1, 2
a, b = b, a
print(a)  # 2
print(b)  # 1
```

> **案例:** 冒泡排序
>
> 冒泡排序（冒泡排序）也是一种简单的简单列访问排序。它重复地走过去要排序的数，一次比较两个元素，如果他们的顺序错误让他们交换过来。走访数工作列的是重复地进行到那时已经没有再需要交换了，该算法的名字会因为越过元素会来交换慢慢地“浮”到数列的排序。
>
> 需求: 任意给定一组序列数字, 比如 **[3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]**, 将序列从小到大排列。（不能使用python函数解决）

```
"""
        需求：任意给定一组序列数字，
        比如 [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]，
        将序列从小到大排列。（不能使用python函数解决）
```

```
"""

def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr

print(bubbleSort([3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]))
# 结果：[2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]
```

2、递归

2.1 递归的应用场景

递归是一种编程思想，应用场景：

1. 在我们日常开发中，如果要遍历一个文件夹下面所有的文件，通常会使用递归来实现；
2. 在后续的算法课程中，很多算法都离不开递归，例如：快速排序。

2.1.1 递归的特点

- 函数内部自己调用自己
- 必须有出口

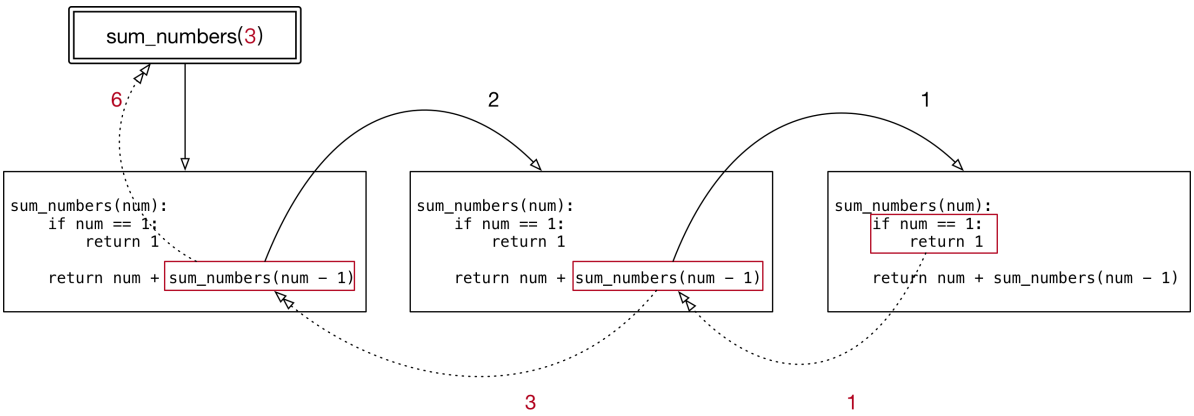
2.2 应用：数字累加求和

- 代码

```
# 3 + 2 + 1
def sum_numbers(num):
    if num == 1:
        return 1
    return num + sum_numbers(num-1)


sum_result = sum_numbers(3)
# 输出结果为6
print(sum_result)
```

- 执行结果



> 案例： 求阶乘， 给定一个数， 求1到这个数的阶乘结果。

3、lambda 表达式

3.1 lambda的应用场景

如果一个函数有一个返回值，并且只有一句代码，可以使用 lambda简化。

3.2 lambda语法

```
lambda 参数列表 ： 表达式
```

> 注意

- lambda表达式的参数可有可无，函数的参数在lambda表达式中完全适用。
- lambda表达式能接收任何数量的参数但只能返回一个表达式的值。

快速入门

```python
# 函数
def fn1():
    return 200


print(fn1)
print(fn1())


# lambda表达式
fn2 = lambda: 100
print(fn2)
print(fn2())
```

> 注意：直接打印lambda表达式，输出的是此lambda的内存地址

3.3 示例：计算a + b

3.3.1 函数实现

```
def add(a, b):
    return a + b


result = add(1, 2)
print(result)
```

> 思考：需求简单，是否代码多？

3.3.2 lambda实现

```
fn1 = lambda a, b: a + b
print(fn1(1, 2))
```

3.4 lambda的参数形式

3.4.1.无参数

```
fn1 = lambda: 100
print(fn1())
```

3.4.2.一个参数

```
fn1 = lambda a: a
print(fn1('hello world'))
```

3.4.3.默认参数

```
fn1 = lambda a, b, c=100: a + b + c
print(fn1(10, 20))
```

3.4.4.可变参数：*args

```
fn1 = lambda *args: args
print(fn1(10, 20, 30))
```

> 注意：这里的可变参数传入到lambda之后，返回值为元组。

3.4.5.可变参数：**kwargs

```
fn1 = lambda **kwargs: kwargs
print(fn1(name='python', age=20))
```

3.5 lambda的应用

3.5.1. 带判断的lambda

```
fn1 = lambda a, b: a if a > b else b
print(fn1(1000, 500))
```

3.5.2. 列表数据按字典key的值排序

```
students = [
    {'name': 'TOM', 'age': 20},
    {'name': 'ROSE', 'age': 19},
    {'name': 'Jack', 'age': 22}
]

# 按name值升序排列
students.sort(key=lambda x: x['name'])
print(students)

# 按name值降序排列
students.sort(key=lambda x: x['name'], reverse=True)
print(students)

# 按age值升序排列
students.sort(key=lambda x: x['age'])
print(students)
```

4、高阶函数

==把函数作为参数传入==，这样的函数称为高阶函数，高阶函数是函数式编程的体现。函数式编程就是指这种高度抽象的编程范式。

4.1 体验高阶函数

在Python中，`abs()`函数可以完成对数字求绝对值计算。

```
abs(-10)  # 10
```

`round()`函数可以完成对数字的四舍五入计算。

```
round(1.2)  # 1
round(1.9)  # 2
```

需求：任意两个数字，按照指定要求整理数字后再进行求和计算。

- 方法1

```
def add_num(a, b):
    return abs(a) + abs(b)


result = add_num(-1, 2)
print(result)  # 3
```

- 方法2

```
def sum_num(a, b, f):
    return f(a) + f(b)


result = sum_num(-1, 2, abs)
print(result)  # 3
```

> 注意：两种方法对比之后，发现，方法2的代码会更加简洁，函数灵活性更高。

函数式编程大量使用函数，减少了代码的重复，因此程序比较短，开发速度较快。

4.2 内置高阶函数

4.2.1 map()

map(func, lst)，将传入的函数变量func作用到lst变量的每个元素中，并将结果组成新的列表(Python2)/迭代器(Python3)返回。

需求：计算`list1`序列中各个数字的2次方。

```
list1 = [1, 2, 3, 4, 5]


def func(x):
    return x ** 2


result = map(func, list1)

print(result)  # <map object at 0x0000013769653198>
print(list(result))  # [1, 4, 9, 16, 25]
```

4.2.2 zip()

zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表。

> 如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同

```
a = [1,2,3]
b = [4,5,6]
c = [4,5,6,7,8]

result = zip(a,b)  # 打包为元组的列表
# 结果：[(1, 4), (2, 5), (3, 6)]

result = zip(a,c)  # 元素个数与最短的列表一致
# 结果：[(1, 4), (2, 5), (3, 6)]
```

4.2.3 reduce()

reduce() 函数会对参数序列中元素进行累积。

reduce(func，lst)，其中func必须有两个参数。每次func计算的结果继续和序列的下一个元素做累积计算。

> 注意：reduce()传入的参数func必须接收2个参数。

需求：计算`list1`序列中各个数字的累加和。

```
import functools

list1 = [1, 2, 3, 4, 5]


def func(a, b):
    return a + b


result = functools.reduce(func, list1)

print(result)  # 15
```

4.2.4 filter()

filter(func, lst)函数用于过滤序列, 过滤掉不符合条件的元素, 返回一个 filter 对象。如果要转换为列表, 可以使用 list() 来转换。

```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


def func(x):
    return x % 2 == 0


result = filter(func, list1)

print(result)  # <filter object at 0x0000017AF9DC3198>
print(list(result))  # [2, 4, 6, 8, 10]
```

5、总结

- 递归
  - 函数内部自己调用自己
  - 必须有出口
- lambda
  - 语法

```
lambda 参数列表: 表达式
```

  - lambda的参数形式
    - 无参数

```
lambda: 表达式
```

    - 一个参数

```
lambda 参数: 表达式
```

    - 默认参数

```
lambda key=value: 表达式
```

    - 不定长位置参数

```
lambda *args: 表达式
```

    - 不定长关键字参数

```
lambda **kwargs: 表达式
```

- 高阶函数
  - 作用：把函数作为参数传入，化简代码
  - 内置高阶函数
    - map()
    - reduce()
    - filter()