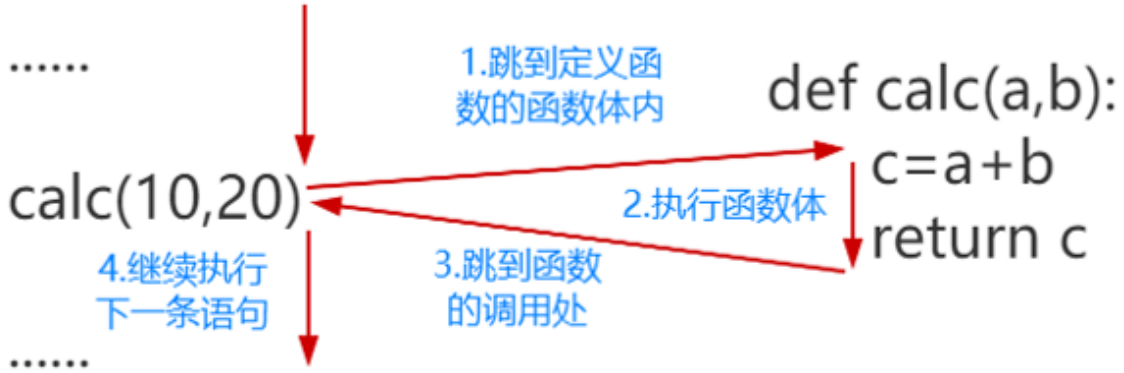
函数，模块化编程

- 什么是函数：函数就是执行特定任和以完成特定功能的一段代码
- 为什么需要函数
  - 复用代码
  - 隐藏实现细节
  - 提高可维护性
  - 提高可读性便于调试

函数的创建

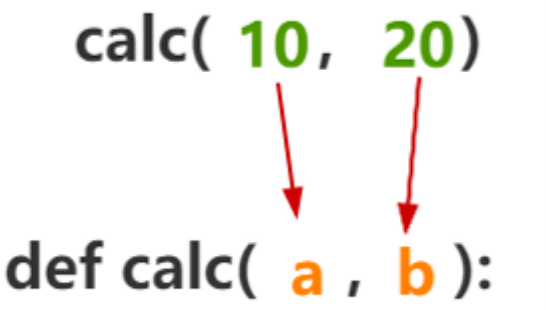
```
def    函数名（[输入参数]）：
    函数体
    [return xxx]
```

- 函数在`执行时`才进行调用

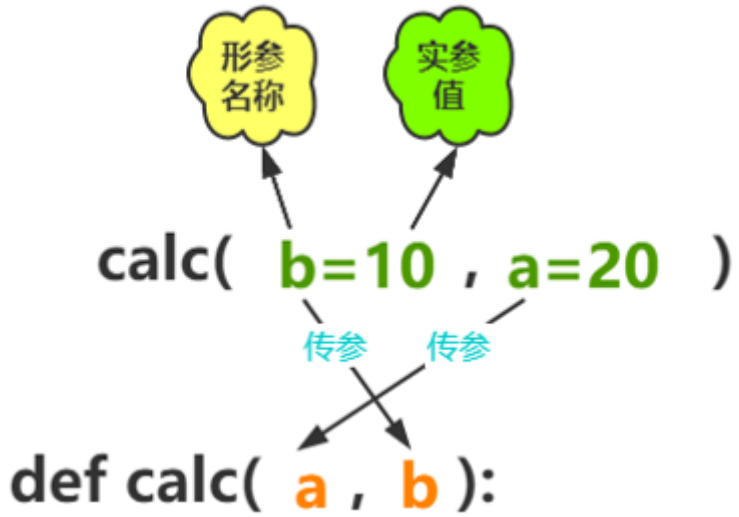


参数传递

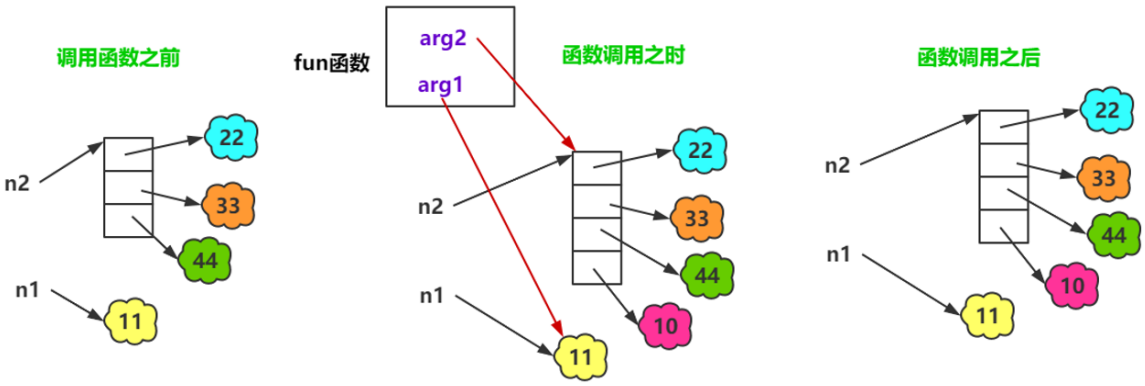
- 位置实参：根据形参对应的位置进行实参传递



- 关键字实参：根据形参名称进行实参传递



- 函数调用的参数传递内存分析



函数的返回值

- 在程序开发中，有时候，会希望 一个函数执行结束后，告诉调用者一个结果，以便调用者针对具体的结果做后续的处理
- 返回值是函数完成工作后，最后给调用者的一个结果
- 在函数中使用 `return` 关键字可以返回结果，一旦返回，函数终止
- 调用函数一方，可以使用变量来接收函数的返回结果
- 函数返回多个值时，结果为`元组`

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

ret1 = add(1, 2)
ret2 = add("hello", "world")

print("ret1 =", ret1)
print("ret2 =", ret2)
```

局部变量和全局变量

局部变量

- **局部变量**是在**函数内部**定义的变量，只能在函数内部使用
- 函数执行结束后，函数内部的局部变量，会被系统回收
- 不同的函数，可以定义相同的名字的局部变量，但是**彼此之间**不会产生影响

```
def f1():

    num = 10
    print("函数1中的num =", num)

    num = 20
    print("函数1中的num改变后 =", num)


def f2():

    num = 30
    print("函数2中num =", num)

f1()
f2()
```

局部变量的生命周期

- 所谓**生命周期**就是变量从**被创建**到**被系统回收**的过程
- **局部变量**在**函数执行时**才会被创建
- 函数执行结束后**局部变量**被系统回收

全局变量

全局变量是在**函数外部定义**的变量，所有函数内部都可以使用这个变量

```
num = 10

def f1():

    print("函数1中num =", num)

def f2():

    print("函数2中num =", num)

f1()
f2()

```

注意：函数执行时，需要处理变量时会：

- 首先查找**函数内部**是否存在**指定名称**的局部变量，如果有，直接使用。
- 如果没有，查找**函数外部**是否存在**指定名称**的全局变量，如果有，直接使用。
- 如果还没有，程序报错！

- 函数不能直接修改全局变量的引用。

在函数内部修改全局变量的值：如果在函数中需要修改全局变量，需要使用 `global` 进行声明

```
num = 10

def f1():

    global num
    num = 20

    print("函数1中num =", num)

def f2():

    global num
    num = 30

    print("函数2中num =", num)

```

```
num = 10

def f1():

    num += 20
    print("函数1中num =", num)

def f2():

    num += 30
    print("函数2中num =", num)

f1()
f2()
```

缺省参数

- 定义函数时，可以给**某个参数**指定一个默认值，具有默认值的参数就叫做 缺省参数。
- 调用函数时，如果没有传入**缺省参数** 的值，则在函数内部使用定义函数时指定的参数默认值。
- 函数的缺省参数，将常见的值设置为参数的缺省值，从而简化函数的调用。

```
def f(m, n, k=1):

    return (m+n) * k

print(f(1, 2))
print(f(1, 2, 3))
```

多值参数

- 使用`*`定义个数可变的位置形参，结果为一个`元组`

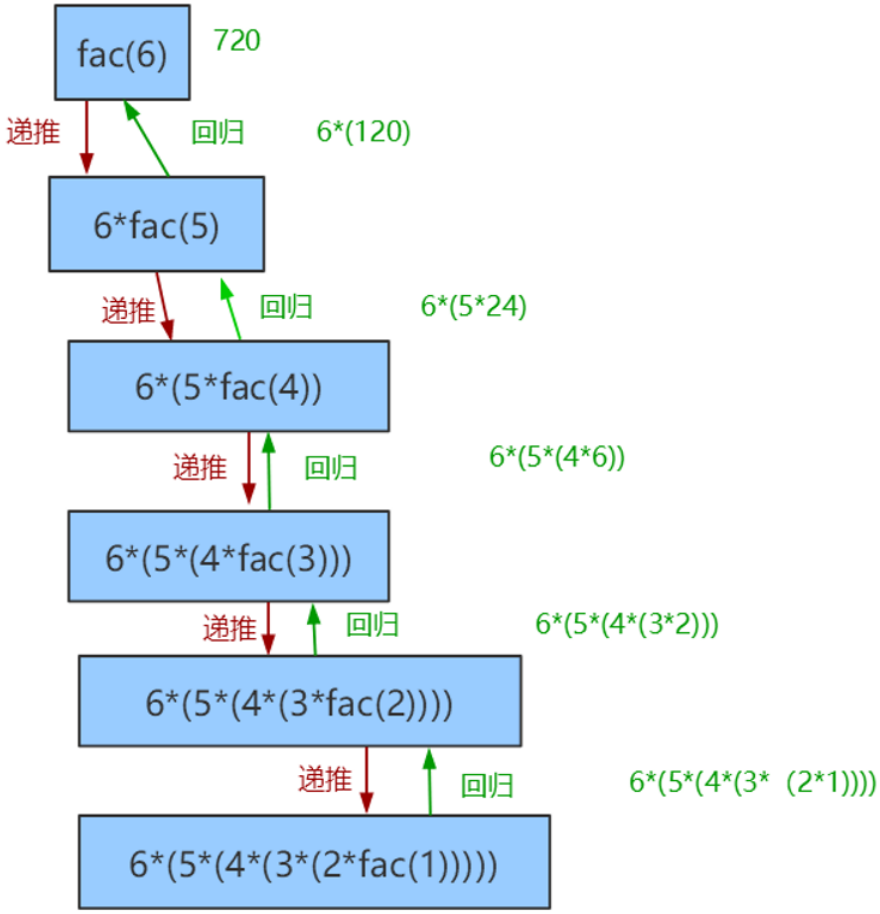
```
def f1(*args):

    for i in args:
        print(i)

f1(1, 2, 3)
```

- 使用`**`定义个数可变的关键字形参，结果为一个`字典`

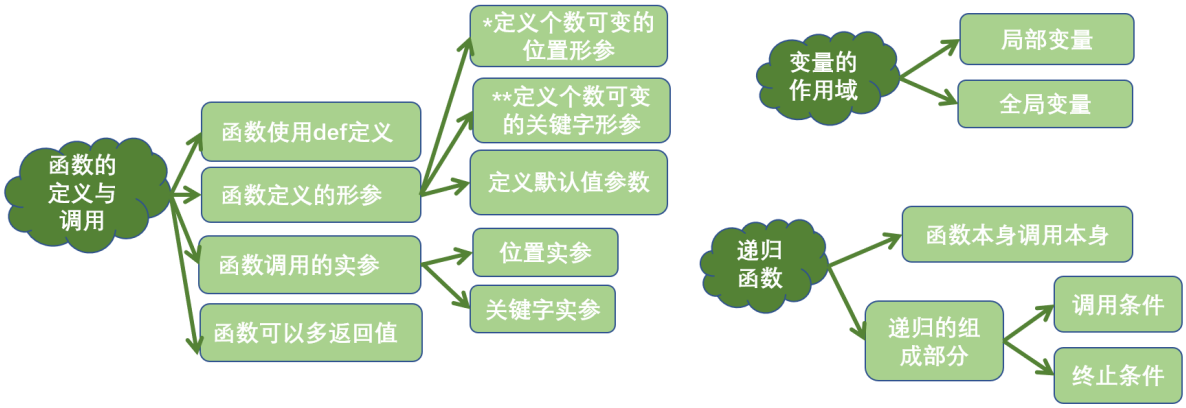
```
def f(**kwargs):
    print(kwargs)

f(a=1, b=2, name="xz")

结果：
{'a': 1, 'b': 2, 'name': 'xz'}
```

递归算法

- 递归函数：如果在一个函数的函数体内调用了该函数本身，这个函数就称为`递归函数`。
- 递归的调用过程
  - 每递归调用一次函数，都会在栈内存分配一个栈帧
  - 每执行完一次函数，都会释放相应的空间
- 递归的优缺点
  - 缺点：占用内存多，效率低下
  - 优点：思路和代码简单



总结



函数进阶

函数的嵌套

```python
def f1():

    print("我在f1函数中")

def f2():

    print("我在f2函数中")

    f1()

f2()
```

```python
def wrapper():

    print("我在外层")

    def inner():

        print("我在内层")
        print("内层函数结束")

    inner()

    print("外层函数结束")

wrapper()
```

```python
def wrapper():

    print("我在外层")

    num = 1

    def inner():

        nonlocal num

        num += 1


        print("我在内层")
        print("内层函数调用时num =", num)
        print("内层函数结束\n")

    print("内层函数调用之前num =", num)
    inner()

    print("外层函数结束")

wrapper()
```

函数名的本质

> 函数名的本质是一个指针

```
def f1():

    print("hello")

print(f1)

结果：
<function f1 at 0x10b0c67a0>
```

函数名赋值

```
def f1():

    print("hello")

print(f1)

f2 = f1
f3 = f2

f3()
```

> 高阶函数：一个函数可以作为参数传给另外一个函数，或者一个函数的返回值为另外一个函数（满足其一则为高阶函数。

```
def f1():

    print('我在f1中')


def f2():
    print('我在f2中')

def f3(f):

    f()

def f4(f):

    print('我是f4')

    return f

```

```
f3(f1)
f3(f2)

ret_func = f4(f2)
ret_func()
```

函数闭包

- 闭包作用，**保证数据安全**
- 内层函数对外层函数**非全局变量的引用**就会形成闭包
- 被引用的非全局变量也称**自由变量**，这个自由变量会与内层函数产生一个绑定关系
- 自由变量**不会在内存中消失**

```
def wrapper():

    print('我在外层')

    def inner():

        print('我在内层')
        print('内层函数结束')

    inner()

    print('外层函数结束')

wrapper()
```

```
def wrapper():

    print('我在外层')
    num = 1

    def inner():

        nonlocal num
        num += 1

        print('我在内层')
        print('内层函数调用时：num =', num)
        print('内层函数结束')

    print('内层函数调用之前：num =', num)
    inner()
    print('内层函数调用之后：num =', num)

    print('外层函数结束')

wrapper()
```

例：求比特币的平均收盘价，6000美元、7000美元、8000美元、9000美元、10000美元.....

```
li = []

def average(value):

    li.append(value)

    return sum(li) / len(li)

print(average(6000))
print(average(7000))
print(average(8000))
```

```
def average():

    li = []

    def inner(value):

        li.append(value)

        return sum(li) / len(li)

    return inner

avg = average()
print(avg(6000))
print(avg(7000))
print(avg(8000))
```

装饰器详解

装饰器，就是装修、装饰的意思，但是，不改变原有的程序功能。比如，我装修一个房子，如果不隔音，我在墙上加一层隔音板，却不能把墙拆了，换成隔音材质。

而程序中也是一样，不会对原来的函数造成改变，还要增添新的功能，调用函数时的接口没有变化。

比如，我们要在函数的基础上，增加一个程序效率检测功能，也就是记录函数执行的时间。

方案1

```python
import time

def index():

    time.sleep(2)

start_time = time.time()

index()

end_time = time.time()

print('程序运行%.3f秒' % (end_time - start_time))
```

方案2

```python
import time

def index():

    time.sleep(2)

def calc_time():

    start_time = time.time()
    index()
    end_time = time.time()

    print('程序运行%.3f秒' % (end_time - start_time))

calc_time()
```

方案3

```python
import time

def index():

    time.sleep(2)

def calc_time(f):

    start_time = time.time()
    f()
    end_time = time.time()

    print('程序运行%.3f秒' % (end_time - start_time))

calc_time(index)
```

语法糖

```python
import time

def index():

    time.sleep(2)

def calc_time(f):

    def inner():

        start_time = time.time()
        f()
        end_time = time.time()

        print('程序运行%.3f秒' % (end_time - start_time))

    return inner

# index = calc_time(index)
# index()

# 语法糖
@calc_time
def f():
    time.sleep(1.2)

f()
# f = calc_time(f)
# f()
```

带返回值的装饰器

```python
import time

def calc_time(f):

    def inner():

        start_time = time.time()
        ret = f()
        end_time = time.time()

        print('程序运行%.3f秒' % (end_time - start_time))

        return ret

    return inner

@calc_time
def index():
    time.sleep(2)
    return 'index'
```

```
21
22 print(index())
```

带参数的装饰器

```
1  import time
2
3  def calc_time(f):
4
5      def inner(*args, **kwargs):
6
7          start_time = time.time()
8          ret = f(*args, **kwargs)
9          end_time = time.time()
10
11         print('程序运行%.3f秒' % (end_time - start_time))
12
13         return ret
14
15     return inner
16
17 @calc_time
18 def add(a, b, c):
19     time.sleep(2)
20     return a + b + c
21
22 print(add(1, 2, 3))
```

生成器

函数生成器

```
1  def f():
2
3      a = 1
4      yield a
5
6      b = 2
7      yield b
8
9      c = 3
10
11     yield c
12
13 print(f())
14
15 g = f()
16
17 print(g.__next__())
18 print(g.__next__())
19 print(next(g))
20 # print(next(g))
```

```python
def get_data():

    for i in range(1, 10000):
        yield i

d = get_data()

for i in range(1, 10):
    print(next(d), end=' ')

print()

for i in range(30, 40):
    print(next(d), end=' ')
```

send方法

- send和next都可以让生成器对应的yield向下执行一次
- 第一次获取yield值只能用next不能用send 或者用send(None)
- send可以给上一个yield置传递值

```python
def f():

    a = yield 1
    print('a =', a)

    b = yield a
    print('b =', b)

    c = yield b

ret = f()

print(next(ret))
print(ret.send('hahaha'))
print(ret.send('xxx'))
```

生成器表达式

> 列表推导式比较耗内存,所有数据一次性加载到内存。而生成器表达式遵循迭代器协议，逐个产生元素

```python
g_li = (i for i in range(10))
print(g_li)

for i in g_li:
    print(i)
```

匿名函数

> 语法：
>
> 函数名 = lambda 参数: 返回值

- 匿名函数并不是没有名字，函数的名字就是设置的变量
- 匿名函数只有一行，逻辑结束后直接返回数据

```
f1 = lambda m, n : m + n
print(f1(1, 2))

f2 = lambda m, n : m if m > n else n
print(f2(6, 9))
```

内置函数

zip()方法

- 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对象
- 节约内存
- 可以使用 list() 转换来输出列表
- 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，利用 * 号操作符，可以将元组解压为列表

```
a = [1, 2]
b = [4, 5, 6]
c = [7, 8, 9, 10]

z = zip(a, b, c)
print(z)

print(list(z))

m, n, k = zip(*zip(a, b, c))
print(m, n, k)
```

filter()方法

- 用于过滤序列，过滤掉不符合条件的元素，返回一个迭代器对象，如果要转换为列表，可以使用 list() 来转换
- 该接收两个参数，第一个为函数，第二个为序列，序列的每个元素作为参数传递给函数进行判，然后返回 True 或 False，最后将返回 True 的元素放到新列表中

```python
def is_odd(x):
    return x % 2

print(list(filter(is_odd, list(range(10)))))
```

map()方法

- 会根据提供的函数对指定序列做映射
- 第一个参数 function 以参数序列中的每一个元素调用 function 函数，返回包含每次 function 函数返回值的新列表

```python
def square(x):

    return x ** 2

print(map(square, [1, 2, 3, 4, 5]))
print(list(map(square, [1, 2, 3, 4, 5])))

print(list(map(lambda x: x**2, [1, 2, 3, 4, 5])))
print(list(map(lambda m, n: m+n, [1, 2, 3, 4, 5], [6, 7, 8, 9, 10])))
```

reduce()方法

- 对参数序列中元素进行累积操作
- 函数将一个数据集合（链表，元组等）中的所有数据进行下列操作：用传给 reduce 中的函数 function（有两个参数）先对集合中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 function 函数运算，最后得到一个结果

```python
from functools import reduce

print(reduce(lambda m, n : m + n, list(range(1, 6))))
```