函数

自定义函数

格式：　def func_name([arg1 [, arg2, ... argN]]):
　　func_body

函数调用

- 形参： 函数中声明的参数；实参： 调用时传入的参数

```
def plus(num):
    print(num + 1)

plus(1)  # 函数调用
func = plus  # func和plus指向同一个内存地址
func(1)  # 函数调用

# 定义好的函数可重复使用
plus(2)
func(2)
```

不带参数

```
# 定义一个不带参数的函数，输出一个由*号组成的3层三角形
def trigon():
    for i in range(1, 6, 2):
        print(("*" * i).center(5))

# 调用函数
trigon()
```

带参数

```
"""
定义一个带参数的函数，
sign: 构成三角形的标志，为单个字符
layers: 三角形的层数，为int类型
"""
def trigon(sign, layers):
    width = 2 * layers - 1  # 最底层的宽度
    for i in range(1, width + 1, 2):
        print((sign * i).center(width))

# 调用函数
trigon("*", 3)
```

return 用法

- return 后面可以跟单个对象，多个对象，表达式，不跟
- return 把后面跟的值返回给调用方，并结束对应的函数

```
def add(left, right):
```

```
    res = left + right
    return res  # 返回单个对象

result = add(3, 4)
print(result)



def add(left, right):
    return left + right  # 返回表达式

result = add(3, 4)
print(result)



def add(left, right):
    res1 = left + right
    res2 = left * right
    return res1, res2  # 返回多个对象，把result1，
result2作为一个元组返回

# def add(left, right):
#     res = left + right
#     return res,  # 注意：这也是元组 (res,)

result = add(4, 3)
print(result)
res1, res2 = add(4, 3)  # 解包（序列赋值）
print(res1, res2)



def func():
```

```
    return

res = func()  # return后面什么也不跟，相当于return None，所以打印输出None
print(res)



def func():
    print("hello")
#  return None  函数在没有return时，默认 return None

print(func())  # 注意输出顺序：先输出"hello"，再输出None
```

```
def func():
    for i in range(5):
        print(i)
        return

print(func())


def func():
    for i in range(5):
        print(i)
        break

print(func())



def func():
```

```
    for i in range(5):
        print(i)
        return
    print("ending...")

print(func())


def func():
    for i in range(5):
        print(i)
        break
    print("ending...")

print(func())
```

文档注释

```
def my_abs(x):
    """Return the absolute value of the
argument."""
    return x if x > 0 else -x


print(my_abs(-3))

# 文档注释存放在 __doc__ 属性中
print(my_abs.__doc__)
print(abs.__doc__)
help(my_abs)
```

```
def my_divmod(x, y):
    """
    Return the tuple (x//y, x%y). Invariant:
div*y + mod == x.
    x: number (except complex)
    y: number (except complex)
    """
    div = x // y
    mod = x % y
    return div, mod


print(my_divmod.__doc__)
help(my_divmod)
```

help([object])

- 启动内置的帮助系统
- 如果没有实参，解释器控制台里会启动交互式帮助系统
- 如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息
- 如果实参是其他任意对象，则会生成该对象的帮助页

```
help()  # 启动交互式帮助系统
help("keywords")  # 查看关键字
help(list)  # 生成list的帮助页
```

abs(x)

- x：可以是整数，浮点数，布尔型，复数。

- 返回一个数的绝对值，如果参数是一个复数，则返回它的模。

```
print(abs(-6))  # 6，参数是整数
print(abs(6.9))   # 6.9，参数是浮点数
print(abs(3+4j))  # 5，参数是复数
```

divmod(a, b)

- a：数字（非复数）
- b：数字（非复数）
- 返回一个包含商和余数的元组 (a // b, a % b)

```
print(divmod(7, 3))  # (2, 1)
print(7 // 3, 7 % 3)

print(divmod(-9, 2))  # (-5, 1)
print(-9 // 2, -9 % 2)
```

max(iterable, *[, key, default]) / max(arg1, arg2, *args[, key])

- 返回给定参数的最大值

```
print(max([1, 2, 3]))  # 传入非空 iterable，返回其中的最大值，输出 3
print(max(1, 2, 3, 4))  # 传入多个数值也行，输出 4
print(max("1", "2", "3", "10"))  # 传入数字型字符串也行，不过是按照第一个字符的数值大小来比较的，输出 "3"
print(max([], default=666))  # 传入空 iterable，返回default指定的值，不指定则报错，输出 666
print(max([1, 2, -3], key=abs))  # key参数指定函数，iterable每个元素应用该函数，再执行max，输出 -3
```

min(iterable, *[, key, default]) / min(arg1, arg2, *args[,key])

- 返回给定参数的最小值

```
print(min([1, 2, 3]))  # 传入非空 iterable，返回其中的最小值，输出 1
print(min(1, 2, 3, 4))  # 传入多个数值也行，输出 1
print(min("2", "3", "10"))  # 传入数字型字符串也行，不过是按照第一个字符的数值大小来比较的，输出 "10"
print(min([], default=666))  # 传入空 iterable，返回default指定的值，不指定则报错，输出 666
print(min([1, 2, -3], key=abs))  # key参数指定函数，iterable每个元素应用该函数，再执行min，输出 1
```

pow(base, exp[, mod])

- 返回 base 的 exp 次幂；如果 mod 存在，则返回 base 的 exp 次幂对 mod 取余

```
print(pow(-2, 3))  # -2**3 = -8
print(pow(-2, 3, 3))  # -2**3 % 3 = 1
```

round(number [, ndigits])

- number：数字
- ndigits：保留的小数点位数
- 返回 number 四舍五入到小数点后 ndigits 位精度的值，如果 ndigits 被省略或为 None，则返回值为整数，否则返回值和 number 类型相同

```
# 精确到整数位，距离 0 和 1 相同，则选择偶数0，且省略了ndigits，所以结果为整数 0
print(round(0.5))
```

```
# 精确到整数位，距离 -1 和 0 相同，则选择偶数0，但ndigits没有省略也没有为None，所以结果类型和number类型一样，即浮点型 -0.0
print(round(-0.5, ndigits=0))
# 精确到整数位，距离 -2 和 -1 相同，则选择偶数-2，且ndigits为None，所以结果为整数 -2
print(round(-1.5, ndigits=None))
# 精确到整数位，距离 -2 和 -1 相同，则选择偶数-2，但ndigits没有省略也没有为None，所以结果类型和number类型一样，即浮点型 -2.0
print(round(-1.5, ndigits=0))


# 精确到小数点后两位，根据距离相同选择偶数的规则，预期结果应该是2.68，但是结果却是2.67
# 这跟浮点数的精度有关。我们知道在机器中浮点数不一定能精确表达，因为换算成一串 1 和 0 后可能是无限位数的，机器已经做出了截断处理。
# 那么在机器中保存的2.675这个数字可能就比实际数字要小那么一点点。这一点点就导致了它离 2.67 要更近一点点，所以保留两位小数时就近似到了 2.67。
print(round(2.675, 2))

# 精确到小数点后两位，根据距离相同选择偶数的规则，预期结果应该是2.66，但是结果却是2.67。和上面同理，机器中保存的2.665可能比实际大一点点
print(round(2.665, 2))
```

sum(iterable, /, start=0)

- iterable：元素必需为数字类型的可迭代对象
- start：累加的起始数字，默认为0
- 从 start 开始自左向右对 iterable 的项求和并返回总计值

```
print(sum([1, 2, 3], start=100))  # 100 + 1 + 2
+ 3 = 106
print(sum((1, 2, 3), start=100))  # 106
print(sum({1, 2, 3}, start=100))  # 106
print(sum({1: "name", 2: "age", 3: "address"},
start=100))  # 106
```

类型标注

- Python 对标注类型并不强制，不按照标注类型也不会报错
- 类型标注主要被用于第三方工具，比如类型检查器、集成开发环境、静态检查器等
- 标注以字典的形式存放在函数的 __annotations__ 属性中，并且不会影响函数的任何其他部分
- 自从 Python3.5 以来，发布了 typing 包，为类型标注提供了支持

```
def func(a: int, b: str, c: list, d: tuple, e:
dict, f: set) -> tuple:
    return a, b, c, d, e, f

print(func(1, 2, 3, 4, 5, 6))
print(func(1, "2", [3], (4, ), {5: 5}, {6}))
print(func.__annotations__)


from typing import List, Tuple, Dict, Set,
Union, Callable, Iterable, Iterator, Generator


# List[int]：建议参数为列表，且所有元素为int类型
```

```
# Tuple[int, str]：建议参数为元组，且第一个元素为int类型，第二个元素为str类型
# Dict[str, int]：建议参数为字典，且所有键为str类型，值为int类型
# Set[str]：建议参数为集合，且所有元素为str类型
def func(a: int, b: str, c: List[int], d: Tuple[int, str], e: Dict[str, int], f: Set[str]) -> Tuple:
    return a, b, c, d, e, f

print(func(1, 2, 3, 4, 5, 6))
print(func(1, "2", [3, 4, 5], (4, "5"), {"5": 4}, {"6"}))


# 嵌套
def func(a: Dict[str, List[Tuple[int, int, int]]]):
    pass


# 建议多种类型
def func(a: Union[str, list, set]):
    pass

func({})
func([1])
func("1")
func({1})


# Callable[[int], int] 建议参数为函数，且只有一个为int类型的参数，返回int类型
```

```
def my_sort(a: list, f: Callable[[int], int], c:
bool):
    return sorted(a, key=f, reverse=c)

print(my_sort([1, -3, 2], abs, True))
```

参数传递

- 传不可变对象 & 传可变对象

```
def func(b):
    print(id(a), a)
    print(id(b), b)
    b = 888
    print(id(a), a)
    print(id(b), b)

a = 999
func(a)



def func(b):
    print(id(a), a)
    print(id(b), b)
    b = [999, 888, 777]
    print(id(a), a)
    print(id(b), b)

a = [999, 777]
func(a)
```

```
def func(b):
    print(id(a), a)
    print(id(b), b)
    b.insert(1, 888)
    print(id(a), a)
    print(id(b), b)

a = [999, 777]
func(a)
```

参数分类

必需参数

- 必须传参, 否则报错（实参按照形参的对应位置顺序传入，所以通常也叫：位置参数）

```
def func(a, b):
    print(a - b)

func(4, 3)  # 1
func(3, 4)  # -1
```

关键字参数

- 允许实参和形参顺序不一致，因为是通过关键字确定传入的值（位置参数不能放在关键字参数的后面）

```
def info(name, age):
    print('姓名:', name)
    print('年龄:', age)

info(age=18, name='小明')
info(name='小明', age=18)
```

默认参数

- 默认参数不能写在必需参数的前面，不能写在**kwargs参数的后面
- 调用函数时，如果该参数没有指定实参，则会使用默认值

```
def info(name, age=18):
    print("姓名:", name)
    print("年龄:", age)

info("小明")
info("小明", age=9)
```

不定长参数

- *args：将参数打包成元组给函数体调用，没有值传给它，就是个空元组

```
def func(a, b, *args):
    print(a, b, args)

"""
a,b是必需参数，分别对应1,2
*args为不定长参数，没有值传给它，就是个空元组
"""
func(1, 2)



"""
a,b是必需参数，分别对应1,2
*args为不定长参数，把剩下的3,4打包成元组
"""
func(1, 2, 3, 4)
```

- **kwargs：将参数打包成字典给函数体调用，没有值传给它，就是个空字典

```
def func(a, b, **kwargs):
    print(a, b, kwargs)

"""
a,b是必需参数，分别对应1,2
**kwargs为不定长参数，没有值传给它，就是个空字典
"""
func(1, 2)

"""
a,b是必需参数，分别对应1,2
**kwargs为不定长参数，把关键字形式的参数打包成字典
"""
func(1, 2, name="Tom", age=18)
```

- 参数顺序固定：当必需参数，*arg参数，**kwargs参数同时存在时，它们的顺序是固定的，否则报错（必需参数 -> *arg参数 -> **kwargs参数）

```
# 先是a,b, 再是*args, 最后**kwargs
def func(a, b, *args, **kwargs):
    print(a, b, args, kwargs)

func(1, 2, 3, 4, name="Tom", age=18)
```

特殊参数

- 默认情况下，函数的参数传递形式可以是位置参数或是显式的关键字参数
- 也可以用 `/` 和 `*` 来限制参数的传递形式
- 其中 `/` 为仅限位置参数，限制在它之前的形参必须以位置参数的形式传入，而不能用关键字参数
- 其中 `*` 为仅限关键字参数，限制在它之后的形参必须以关键字参数的形式传入
- 这两个特殊参数只是为了限制参数的传递形式，不需要为它们传入实参

```
def func(pos1, pos2, /, pos_or_kwd, *, kwd1,
kwd2):
    pass


func(1, 2, 3, kwd1=4, kwd2=5)
```

匿名函数

- 格式： lambda [arg1 [, arg2, ... argN]] : expression
- 匿名函数的参数可以有多个, 但是后面的 expression 只能有一个
- 匿名函数返回值就是 expression 的结果，而不需要 return 语句
- 匿名函数可以在需要函数对象的任何地方使用（如：赋值给变量、作为参数传入其他函数等），因为匿名函数可以作为一个表达式，而不是一个结构化的代码块

```
# 定义了一个没有参数的匿名函数，返回一个字符串
# 用标识符func1指向这个匿名函数的内存地址
func1 = lambda : "It just returns a string"
print(func1())  # 调用匿名函数，输出函数的返回值


# 定义了一个含有三个参数的匿名函数，没有返回值
# 用标识符func2指向这个匿名函数的内存地址
func2 = lambda x, y, z: print(x + y + z)
func2(1, 2, 3)  # 调用匿名函数，传入对应实参


# 把匿名函数作为参数传入其他自定义函数
def call_f1(function):
    print(function())

call_f1(func1)
call_f1(lambda : "It just returns a string")


# 把匿名函数作为参数传入其他自定义函数
def call_f2(function, a, b, c):
    function(a, b, c)
```

```
call_f2(func2, 1, 2, 3)
call_f2(lambda x, y, z: print(x + y + z), 1, 2,
3)


# 把匿名函数作为参数传入其他匿名函数
func2 = lambda x, y, z: print(x + y + z)
func3 = lambda function, a, b, c: function(a, b,
c)
func3(func2, 1, 2, 3)


# 把匿名函数作为参数传入其他内置函数
print(sorted([-4, 2, 3], key=lambda x: -x if x <
0 else x))
```

封包、解包

封包

- 将多个值赋值给一个变量时，Python 会自动将这些值封装成元组，这个特性称之为封包

```
a = 1, 2, 3, 4
print(a)
```

解包

可迭代对象都支持解包

- 赋值过程中的解包

```
# 赋值符号左边变量和右边可迭代对象的元素数量一样，则一一对应赋值
a, b, c, d = (1, 2, 3, 4)
print(a, b, c, d)  # 1 2 3 4

# 当左边和右边数量不一样时，会报错
a, b, c, d = (1, 2, 3, 4, 5)
print(a, b, c, d)  # 报错


"""
可以在其中一个变量前面加一个星号(*)，代表这个变量可接收0个/1个/多个元素，并把它们组成列表来赋值，理解起来类似于不定长参数中的*args。解包过程：先把其他变量根据位置确定对应要赋值的元素，剩下的元素都归带星号(*)的变量，组成列表来赋值
"""
a, b, *c, d = (1, 2, 3, 4, 5)
print(a, b, c, d)  # 1 2 [3, 4] 5

a, b, *c, d, e = (1, 2, 3, 4, 5)
print(a, b, c, d, e)  # 1 2 [3] 4 5

a, b, *c, d, e = (1, 2, 3, 4)
print(a, b, c, d, e)  # 1 2 [] 3 4
```

```
# 不允许多个带星号(*)的变量，因为会有歧义
a, *b, *c, d = (1, 2, 3, 4, 5)
print(a, b, c, d)

# 这种解包的写法是错误的
*a = (1, 2, 3, 4, 5)

# 正确的写法应该是这样
*a, = (1, 2, 3, 4, 5)
print(a)  # [1, 2, 3, 4, 5]

# 通常约定：当变量不需要使用时，可以用下划线命名
*a, _, _ = (1, 2, 3, 4, 5)
print(a)  # [1, 2, 3]
```

- 在可迭代对象前面加一个星号（*），在字典对象前面加双星（**），这种解包方式主要运用在函数传参的过程中

```
a = (1, 2, 3, 4, 5)
print(*a)  # 1 2 3 4 5
print(*(1, 2, 3, 4, 5))  # 同上


"""
思考：下面两个 *a 是一样的吗？
不一样。两个星号(*)的作用是不一样的，第一个*a中的星号代表变量a可以接收0个/1个/多个元素，从而得到a为[1, 2, 3]，而第二个*a中的星号代表解包，对变量a解包，即对[1, 2, 3]解包，即 *[1, 2, 3]
"""
*a, b, c = (1, 2, 3, 4, 5)
print(*a)  # 1 2 3
```

```
# 在可迭代对象前面加一个星号（*），使其以位置参数的形式传入函数
def func(a, b, c, d=None):
    print(a, b, c, d)

tup = (1, 2, 3, 4)
dic = {'name': "Tom", 'age': 18, 'height': 188}
func(*tup)  # 等效于 func(1, 2, 3, 4)
func(*dic)  # 等效于 func('name', 'age', 'height')


# 在字典对象前面加双星（**），使其以关键字参数的形式传入函数
def func(a, b, c, d=None):
    print(a, b, c, d)

dic = {'a': "Tom", 'b': 18, 'c': 188, 'd': True}
func(**dic)  # 等效于 func(a="Tom", b=18, c=188, d=True)


dict(**dic)  # 等效于 dict(a="Tom", b=18, c=188, d=True)
```

命名空间与作用域

思考：下面程序的输出结果是什么？

```
list = [1, 2, 3, 4, 1, 2, 1, 2]
set1 = set(list)
print(list(set1))
```

命名空间

定义：命名空间(Namespace)是一个从名称到对象的映射

实现：大部分命名空间当前由 Python 字典实现（内置命名空间由 builtins 模块实现）

作用：提供了在项目中避免名字冲突的一种方法（各个命名空间是独立的，没有任何关系，所以一个命名空间中不能有重名，但不同的命名空间是可以重名而没有任何影响的）

内置命名空间

- 包含所有 Python 内置对象的名称
- 在解释器启动时创建，持续到解释器终止

```
import builtins

print(dir(builtins))
```

dir([object])

- 不带参数时，返回当前范围内的变量、方法和定义的类型列表
- 带参数时，返回参数的属性、方法列表
- 如果参数包含方法__dir__()，该方法将被调用
- 如果参数不包含__dir__()，该方法将最大限度地收集参数信息

- 返回的列表按字母表排序（按照 ASCII 码）

```
print(dir())
print(dir(list))
```

全局命名空间

- 包含模块中定义的名称，记录了模块的变量、函数、类、其它导入的模块等
- 在模块被读入时创建，持续到解释器退出

局部命名空间

- 包含函数中定义的名称，记录了函数的变量、参数等
- 一个函数的局部命名空间在这个函数被调用时创建，持续到函数结束

```
import random, copy


def func1(arg1, arg2):
    num = 666
    print("func1-globals:\n", globals(),
end="\n\n")  # 返回当前全局命名空间的字典
    print("func1-locals:\n", locals(),
end="\n\n")  # 返回当前局部命名空间的字典
    return num, arg1, arg2


def func2(arg1, arg2):
    num = 777
```

```
    print("func2-globals:\n", globals(),
end="\n\n")  # 返回当前全局命名空间的字典
    print("func2-locals:\n", locals(),
end="\n\n")  # 返回当前局部命名空间的字典
    return num, arg1, arg2


num = 111
func1(222, 333)
func2(444, 555)


# 在全局命名空间下，globals()和locals()返回相同的字典
print(globals())
print(locals())
```

命名空间查找顺序

- 局部命名空间 -> 全局命名空间 -> 内置命名空间

作用域

定义：Python 程序可以直接访问命名空间的正文区域

作用：决定了哪一部分区域可以访问哪个特定的名称

分类：（L - E - G - B 作用域依次增大）

- 局部作用域（Local） - L
- 闭包函数外的函数中（Enclosing） - E
- 全局作用域（Global） - G

- 内建作用域（Built-in） - B

规则：在当前的作用域如果找不到对应名称，则去更大一级的作用域去找，直到最后找不到就会报错

说明：只有模块（module）、类（class）以及函数（def、lambda）才会引入新的作用域

局部作用域

```
def func():
    a = 2  # 局部变量
    b = 3  # 局部变量
    print(a + b)  # 局部作用域可以调用局部变量a,b
    print(d)  # 局部作用域可以调用全局变量d


d = 4  # 全局变量
func()
# print(a, b)  # 全局变量不能调用局部变量
```

闭包函数外的函数中

```
def outer():
    b = 2  # Enclosing变量b,c
    c = a + 3  # Enclosing可以调用全局变量a

    def inner():
        c = 5  # 局部变量c
        print(a)  # 局部作用域可以调用全局变量a
        print(b)  # 局部作用域可以调用Enclosing变量
b
```

```
        print(c)  # 优先调用自己作用域内的变量c，而不调用Enclosing变量c

    return inner()


a = 1  # 全局变量
outer()
```

全局作用域

```
def outer():
    a = c + 2  # Enclosing可以调用全局变量c

    def inner():
        b = c + 3  # 局部作用域可以调用全局变量c
        print(a + b)  # 局部作用域可以调用Enclosing变量a

    return inner()


c = 1  # 全局变量
outer()
print(c)  # 调用全局变量c
```

内建作用域

```
# abs是内置函数、int是内置类，它们都在内建作用域builtins模块中
num1 = abs(-100)
num2 = int(3.141592653)
```

global 和 nonlocal

- 当内部作用域想要给外部作用域的变量重新赋值时，可以用 global 或 nonlocal 关键字声明

```
def outer():
    global a, b  # 声明当前作用域的a,b为全局变量
    a, b, c, d = 3, 4, 5, 6
    print(a, b)

    def inner():
        global a, b  # 声明当前作用域的a,b为全局变量
        nonlocal c, d  # 声明当前作用域的c,d为
Enclosing变量
        a, b, c, d = 7, 8, 9, 0

    inner()
    print(c, d)


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

- 易错情况

```
def outer():

    def inner():
```

```
        """ 解决方案一: 在这里用global声明变量a 或者定义一个局部变量a """
        b = a + 1  # 本身没有错，受到下面这行代码的影响才报错
        """ 解决方案二: 把等号左边的a换成其他变量名 """
        a = a + 1  # 因为等号左边的a属于局部变量，这种写法会导致该作用域的其他a都被解释器判定为局部变量，所以会报错：局部变量a在赋值前被引用
        print(a, b)

    return inner()


a = 1
outer()




def outer():
    lis = [1, 2]

    def inner():
        """ 解决方案一: 在这里用nonlocal声明变量lis 或者定义一个局部变量lis """
        res = lis.append(3)  # 本身没有错，受到下面这行代码的影响才报错
        """ 解决方案二: 把等号左边的lis换成其他变量名 """
        lis = lis.append(3)  # 因为等号左边的lis属于局部变量，这种写法会导致该作用域的其他lis都被解释器判定为局部变量，所以会报错：局部变量lis在赋值前被引用
        print(lis, res)
```

```
    return inner()


outer()
```

常用高阶函数

定义：参数或（和）返回值为其他函数的函数

filter(function, iterable)

- function：函数（function 的必需参数只能有一个），也可以为 None
- iterable：可迭代对象
- 将 iterable 中每个元素作为参数传递给函数，根据函数的返回结果进行判断 True 或 False，将判断为 True 的 iterable 中的元素构建新的迭代器并返回
- 如果 function 为 None，直接判断 iterable 中元素 True 或 False，再返回为 True 的元素构建的新的迭代器

```
# 思考：如果换成 lambda x: print(x-1) 会怎样?
object1 = filter(lambda x: x-1, [1, 2, 3, False,
4])
print(list(object1))

object3 = filter(None, [1, 2, 0, 3, False, 4])
print(list(object3))
```

map(func, *iterables)

- func：函数（func 的必需参数要和 iterables 个数相同）
- iterables：可迭代对象
- 用 iterables 中的每个元素作为函数的参数来调用函数，以迭代器形式返回所有结果
- 当有多个 iterables 对象时，最短的 iterables 耗尽则函数停止

```
def square(a):
    return a**2  # 思考：如果改为 print(a**2) 会怎样?

result = map(square, [1, 2, 3])
print(list(result))

result = map(lambda a: a**2, [1, 2, 3])
print(list(result))

result = list(map(float, ["1", "2", "3"]))
print(result)

# 类似于zip的取元素方式
result = list(map(lambda x, y, z: x+y+z, [1, 2, 3], [3, 2, 1], [1, 3, 2]))
print(result)

# 当有多个 iterables 对象时，最短的 iterables 耗尽则函数停止
result = list(map(lambda x, y, z: x+y+z, [1, 2, 3], [3, 2, 1], [1, 3]))
print(result)
```

reduce(function, iterable[, initial])

- function：函数（function 必需参数只能有两个）
- iterable：可迭代对象
- initial：初始值
- 在 Python2 中 reduce() 是内置函数，而在Python3中 reduce() 函数是在functools模块中的，所以在使用的时候需要先导入 functools 模块
- 在没有指定 initial 参数时，先把 iterable 的前两个元素作为参数调用函数，把这次函数的结果以及iterable 的下一个元素又作为参数再调用函数，以此类推
- 在指定 initial 参数时，先把 initial 值和 iterable 的第一个元素作为参数调用函数，把这次函数的结果以及 iterable 的下一个元素又作为参数再调用函数，以此类推
- 如果 iterable 为空，返回 initial ，此时如果没有指定 initial，则报错
- 如果 iterable 只有一个元素且没有指定 initial，返回该元素

```
from functools import reduce


def add(m, n):
    s = m + n
    return s   # 如果改为 print(s) 会怎样?

# 过程：[(1+2)+3]+4 = 10
result = reduce(add, [1, 2, 3, 4])
print(result)

# 过程：2*[2*(2*5+1)+2]+3 = 51
result = reduce(lambda x, y: 2*x + y, [1, 2, 3],
5)
print(result)

# iterable为空，返回initial
```

```
result = reduce(lambda x, y: 10*x + 2*y, [],
123)
print(result)  # 123

# iterable只有一个元素且没有指定 initial，返回该元素
result = reduce(lambda x, y: 10*x + 2*y, [123])
print(result)  # 123

# 过程：10*2 + 2*123 = 266
result = reduce(lambda x, y: 10*x + 2*y, [123],
2)
print(result)
```

递归函数（了解）

定义：程序调用自身的编程技巧称为递归。

思想：将一个大问题分解成一个个的小问题，然后再从小问题回推出大问题

一般来说，递归函数要满足2个条件：

- 递归边界条件（一般到递归边界则终止当前递归）
- 递归推理（一般是提取重复的子问题，不断向递归边界靠拢或者不断缩小问题规模）

兔子问题

一般而言，兔子在出生两个月后，就有繁殖能力，一对兔子每个月能生出一对小兔子来。如果所有兔子都不死，那么怎么确定第 n 个月有多少对兔子呢？

```
# 循环实现
def func(m):
    m0 = 1
    m1 = 1
    for _ in range(m-1):
        m0, m1 = m1, m0+m1
    return m1

for i in range(10):
    print(func(i))


# 递归实现
def get_rabbits(m):
    if m < 2:
        return 1
    return get_rabbits(m-1) + get_rabbits(m-2)


print(get_rabbits(12))
```

最大递归深度限制

```
def get_rabbits(m):
    if m < 2:
        return 1
    return get_rabbits(m-1) + get_rabbits(m-2)

print(get_rabbits(998))
print(get_rabbits(999))
```

```python
import sys


def get_rabbits(m):
    if m < 2:
        return 1
    return get_rabbits(m-1) + get_rabbits(m-2)


print(sys.getrecursionlimit())  # 返回默认的最大递归深度1000
sys.setrecursionlimit(1500)  # 设置最大递归深度为1500
print(get_rabbits(999))  # 放宽最大递归深度之后不报错了
```

解决递归重复计算问题

当 m 比较大时，比如 m=50，会发现程序计算会变得非常慢，因为递归程序进行了大量的重复计算；要解决递归的重复计算问题，只要把之前已经计算过的数和结果储存起来，后面如果再计算这个数就直接取结果

```python
store={}
def get_rabbits(m):
    if m < 2:
        return 1

    if m in store:
        return store[m]

    result = get_rabbits(m-1) + get_rabbits(m-2)
    store[m] = result
    return result
```

```
for m in range(1000):
    result = get_rabbits(m)
    if m == 999: print(result)
```