Python函数
基本用法

本节重点：

- 函数基本用法
- 局部变量和全局变量
- 常用内置函数
- 递归函数
- 匿名函数

功能需求

1个监控程序，24小时全年无休的监控你们公司网站服务器的系统状况，当cpu＼memory＼disk等指标的使用量超过阀值时即发邮件报警。

```
while True：
    if cpu利用率 > 90%:
        #发送邮件提醒
        连接邮箱服务器
        发送邮件
        关闭连接
    if 硬盘使用空间 > 90%:
        #发送邮件提醒
        连接邮箱服务器
        发送邮件
        关闭连接
    if 内存占用 > 80%:
        #发送邮件提醒
        连接邮箱服务器
        发送邮件
        关闭连接
```

- 代码冗余，重复代码，low
- 简单修改需求，需要同时修改多个地方

问题

提取公因子

合并同类项

解决方法

```
def 发送邮件(内容)
    #发送邮件提醒
    连接邮箱服务器
    发送邮件
    关闭连接

while True：
    if cpu利用率 > 90%:
        发送邮件('CPU报警')
    if 硬盘使用空间 > 90%:
        发送邮件('硬盘报警')
    if 内存占用 > 80%:
        发送邮件('内存报警')
```

函数定义

- 函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。
- 函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

函数作用:

- 减少重复代码
- 使程序变的可扩展
- 使程序变得易维护

使用示例:

```
def sayhi():#函数名
    print("Hello, I'm nobody!")
sayhi() #调用函数
```

```
#下面这段代码
a,b = 5,8
c = a**b
print(c)
```

```
a,b=10,20
def calc(x,y):
    res = x**y
    return res #返回函数执行结果
c = calc(a,b) # 结果赋值给c变量
print(c)
```

函数的规则

- 函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。
- 任何传入参数和自变量必须放在圆括号中间，圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号起始，并且缩进。
- return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

```
def funcname( paras ):
   function_suite
   return [expression]
```

```
def printme( str ):
    print(str)
     return
```

函数的调用

- 定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。
- 这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

```
# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

传入的函数参数

```
# 函数的定义：
def function_name(para1, para2, para3):
#函数名(形参)
    pass
    return return_value
# 函数的使用
function_name(1, 2, 3) #调用函数(实参)
```

```
# 位置参数  and 默认参数
def function_name(para1, para2, para3 = 12):#函数名
    pass
    return return_value
# 函数的使用
function_name(1, 2) #调用函数
```

注意：所有位置参数必须出现在默认参数前，包括函数定义和调用。

```
# 关键字参数（不需要按照顺序传递）
def function_name(para1 = 1, para2 = 2, para3 = 3):
    print(para1, para2, para3)
    return return_value
# 关键参数函数的使用
function_name(para2 = 1, para1 = 12) #调用函数
```

有位置参数时，位置参数必须在关键字参数的前面，但关键字参数之间不存在先后顺序的

代码示例：

```
def function_name(para1, para2, para3):
    print(para1, para2, para3)
    return ":".join([str(para1), str(para2), str(para3)])

a = function_name(1, 2, 3)
print(a)

# 调用方法1：
b = function_name(para1=11,para2=22,para3=21)

# 调用方法2：'
dict = {'para1: 11, 'para2': 22, 'para3':21}
c = function_name(**dict)
print(c)
```

可变参数传入

定义先后顺序：位置参数 > 不定数组*args = 默认参数 > **kwargs

若你的函数在定义时不确定用户想传入多少个参数，就可以使用非固定参数

def stu_register(name,age,*args):

*args 会把多传入的参数变成一个元组形式

print(name,age,args)

stu_register(“Jessie”,32,*(”china”,175.2))

```
def function_name(para1, para2, para3, *args):
    print(para1, para2, para3)
    print(args)
    print(kwargs)
    return ":".join([str(para1), str(para2), str(para3)])


a = function_name(1, 2, 3)
print(a)

list=(11, 22, 33)
b = function_name(para1=11, para2=22, para3=33, *list,)
print(b)
```

若你的函数在定义时不确定用户想传入多少个参数，就可以使用非固定参数

def stu_register(name,age,*args, **kwargs):

**kwargs 会把多传入的参数变成一个dict形式

print(name,age,args)

stu_register(“Jessie",32, **{'province': 'ShanDong'})

```
def function_name(para1, para2, para3, *args, **kwargs):
return ":".join([str(para1), str(para2), str(para3)])


a = function_name(1, 2, 3)
print(a)

list=(11, 22, 33)
dict = {'para1': 11, 'para2': 22, 'para3':21, 'para4':21,
'para5':22}
b = function_name(para1=11, para2=22, para3=33, *list, **dict)
print(b)
```

- 思考：能否传入函数对象？

函数输入参数

```
def foo(*args, **kwargs):
    print('args = ', args)
    print('kwargs = ', kwargs)
    print('------------------')

if __name__ == '__main__':
    foo(1,2,3,4)
    foo(a=1,b=2,c=3)
    foo(1,2,3,4, a=1,b=2,c=3)
    foo('a', 1, None, a=1, b='2', c=3)
```

```
def bar(x,y,z):
    print(x)
    print(y)
    print(z)
bar(*(1,2,3))    #实参一一对应
# bar(*(1,2,3,4))
#会报错只需要3个值，
# 但给定的实参有4个值，不能一一对应
```

```
#*args的运用
def foo (x,*args):
    print(x)
    print(args)

foo(1,2,3,4,5,6,6,'a','b')
#调用函数
```

```
#*args与位置参数和默认参数混用:*args要放到位置参数后面，默认参数要放最后。
def foo(x,y = 1,*args):
    print(x)
    print(y)
    print(args)
foo(1,2,3,4,5,6,7,8,9,10)    #调用函数
```

```
#*args与位置参数和默认参数混用:*args要放到位置参数后面，默认参数要放最后。
def foo(x,*args ,y = 1):
    print(x)
    print(y)
    print(args)
foo(1,2,3,4,5,6,7,8,9,10)    #调用函数
```

思考：

- 传入简单变量，函数内修改是否对外有影响。
- 传入复杂变量（list or dict)，是否能将修改结果传递到外面？

函数返回值

函数外部的代码要想获取函数的执行结果，就可以在函数里用return语句把结果返回

- 函数在执行过程中只要遇到return语句，就会停止执行并返回结果，so 也可以理解为 return 语句代表着函数的结束。
- 如果未在函数中指定return,那这个函数的返回值为None。
- 思考：能否返回函数对象？

课堂练习（1）- 生成人员列表的几种方法

```
# 生成人员列表的两种方法
# 方法1，直接法。
# person_list = [{
#     "name": "张三", "age": 19, "country": "中国",
# },
#     {
#         "name": "李四", "age": 29, "country": "中国",
#     }
#     ,
#     {
#         "name": "王五", "age": 9, "country": "中国",
#     }
# ]
# print(person_list)
```

课堂练习（1）- 生成人员列表的几种方法

```
# 方式2：函数法
# def create_person(name, age, contry):
#     return {"name":name, "age":age, "country": contry}
#
```

```
# 带默认传值的函数，  没有传入，则取默认值。
# def create_person(name, age, contry = '中国'):
#     return {"name":name, "age":age, "country": contry}
```

```
# 但是，函数的调用有区别，关键参数可以无序传值，但不能重复定义
# person_list =[]
# person_list.append(create_person(name = "张三", age = 18, country='中国'))
# person_list.append(create_person(age = 29, name = "李四", country = '中国'))
# person_list.append(create_person(name = "王五", country = '美国', age = 9))
# print(person_list)
```

课堂练习（1）- 生成人员列表的几种方法

```
# 位置参数与关键参数共同传入时，需要先传入位置参数，再传入关键参数！
# def create_person(name, country, age):
#     return {"name":name, "age":age, "country": country}
#

# person_list =[]

# 正确的做法
# person_list.append(create_person("张三", age = 18, country='中国'))
# person_list.append(create_person(name = "王五", country = '美国', age = 9))

# 错误的做法
# person_list.append(create_person(age = 29, "李四", country = '中国'))
# print(person_list)
#
```

课堂练习（1）- 生成人员列表的几种方法

```
# 可变参数传入(元组方式)
# def add(*args):
#     print("args:",args)
#     return sum(args)
#     # sum = 0
#     # for val in args:
#     #     sum += val
#     # return sum

# print("add(1, 2)", add(1, 2))
# print("add(1, 2, 3)", add(1, 2, 3))
# print("add(1, 2, 3, 4)", add(1, 2, 3, 4))
# print("add(1, 2, 3, 4, 5)", add(1, 2, 3, 4, 5))
```

```
# print(add(1,2))
# tup = (1,2)
# print(add(*tup))
```

课堂练习（1）- 生成人员列表的几种方法

```
# 可变参数传入(字典方式)
# def create_person(*args, **kwargs):
#     # print(args)
#     # print(kwargs)
#     if kwargs.get('cc', None):
#         print("this person has cc!!!", kwargs['name'])
#     else:
#         print("this person not has cc!!!", kwargs['name'])
#     return kwargs
#     # return {}
#     # return {"name":name, "age":age, "country": country}
#

# person_list =[]
# # person_list.append({"name" : "张三", "age" : 18, "country":'中国', "red" : True})
# person_list.append(create_person(age = 29, name = "李四", country = '中国'))
# person_list.append(create_person(name = "王五", country = '美国', age = 9, cc = 'tt'))
# print(person_list)
```

课堂练习（1）- 生成人员列表的几种方法

```
# 定义先后顺序：位置参数 > 默认参数 > 不定数组*args  > **kwargs
# def create_person(name, age = 18, *args, **kwargs):
#     print(name)
#     print(*args)
#     print(age)
#     print(kwargs)

# 两种正确的调用方式
# create_person('张三', 1, 2,3, country = '中国', age = 22)
# create_person('张三', 1, 2,3, age = 22, country = '中国')
# 2.用法有问题！
# create_person('张三', 1, 4, 5, 6, age = 22, country = '中国')
```

课堂练习（1）- 生成人员列表的几种方法

用9种方法去生成人员信息的列表，并针对每一个函数，添写一种调用语句。

- create_peason_1(name, age, height, weight , country)
- create_peason _2(name, age, height, weight, country = “china”)
- create_peason _3(name, age, country = “china” , *args)
- create_peason _4(name, age, country = “china” , **kwargs)
- create_peason _5(name, age , country = “china”, *args, **kwargs)
- create_peason _6(name, age, *args, **kwargs)
- create_peason _7(*args , **kwargs)
- create_peason _8(*args)
- create_peason _9(**kwargs)

课堂练习（2）- 几种函数参数的使用

```
#
# def foo(*args, **kwargs):
#     print(args)
#     print(kwargs)
#
#
# foo(1, 2, 3, d=5, c=4)
#
# def foo(*args, **kwargs):
#     print(args)
#     print(kwargs)
# foo(1,2,3, key = 1, 4,5,6,cc = 2)

# def foo(x, *args, y=2):
#     print(x)
#     print(y)
#     print(args)
# foo(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, y=2)  # 调用函数
#
# #*args与位置参数和默认参数混用:*args要放到位置参数后面，默认参数要放最后。
#
```

课堂练习（2）- 几种函数参数的使用

```
# def bar(x,y,z):
#     print(x)
#     print(y)
#     print(z)
#
# # bar(*(1,2,3))
# # <==> 等价于 bar(1,2,3)
# dic={'x':1, 'y':2,'z':3}
# print(dic)
# # print(**dic)
# # bar(**dic)
```

```
# def foo(x, y, z):
#     print(x,y,z)

# tup = 1, 2, 3
# foo(1, 2, 3)
# foo(*(1, 2, 3)) <==> foo(1, 2, 3) <==> foo(*tup)
# print(*tup, sep='<>') #<==>print(1, 2, 3)
# 两种方式是等价的。。。
# print(1,2,3, sep='<>') #<==>print(1, 2, 3)
# foo(*tup)
```

课堂练习（2）- 几种函数参数的使用

```
# def bar(x,y,z):
#     print(x)
#     print(y)
#     print(z)
#
# # bar(*(1,2,3))
# # <==> 等价于 bar(1,2,3)
# dic={'x':1, 'y':2,'z':3}
# print(dic)
# # print(**dic)
# # bar(**dic)
```

```
# def foo(x, y, z):
#     print(x, y, z)

# tup = 1, 2, 3
# foo(1, 2, 3)
# foo(*(1, 2, 3)) <==> foo(1, 2, 3) <==> foo(*tup)
# print(*tup, sep='<>') #<==>print(1, 2, 3)
# 两种方式是等价的。。。
# print(1,2,3, sep='<>') #<==>print(1, 2, 3)
# foo(*tup)
```

课堂练习（2）- 几种函数参数的使用

```
# def foo(*args, **kwargs):
#     print(args)
#     print(kwargs)

# foo(1, 2, 3, a = 1, b = 2, c= 'c')
# foo(1, 2, a = 1, b = 2, c= 'c', d = 12, c = 'aaa')
```

```
# def foo(x, y = 1, *args):
# def foo(x, *args, y=1):
#     print(x)
#     print(y)
#     print(args)

# foo(1, 2, 3, 4, 5, 6, 7, 8, y = 12)

# foo(1, 2, 3, 4, 5, 6, 7, 8)
# foo(1, 2, 3, 4, 5, 6, 7, 8, y = 22)
# foo(1, 2, 3, 4, 5, 6, 7, 8, y = 22, z = 15)

# foo(1, y = 22, 2, 3, 4, 5, 6, 7, 8)
```

课堂练习（2）- 几种函数参数的使用

```
# def foo(x, y = 1, *args, **kwargs):
#     print(x)
#     print(y)
#     print(args)
#     print(kwargs)
# foo(1, x1 = 10, y = 20, z1 = 30)
```

课堂练习（4）-多变量赋值与tuple之间的关系

```
# 多变量赋值与tuple之间的关系, python 理念，简洁、易懂。
# tup = 10, 2
# print(tup)
# a, b = tup
# print(a, b)

# int i = 0;
# int j = 10;
# int k = 20;
# i, j, k = 0, 10, 20
```

全局与局部变量

```
value = 123
def change_name():
    print("after change", value)
change_name()
print("在外面看看value改了么?",value)
```

- 两个value 不是同一个变量， 可以查看id()
- 在函数中定义的变量称为局部变量，在程序的一开始定义的变量称为全局变量。
- 全局变量作用域(即有效范围)是整个程序，局部变量作用域是定义该变量的函数。
- 变量的查找顺序是**局部变量>全局变量**
- 如果在函数里直接修改全局变量，需要增加global声明。

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。

课堂练习（5）-局部变量和全局变量

```
# LEGB
# 变量的引用顺序： 先局部变量 再全局变量，如果局部变量没有，再去找全局变量。
# locals()： 查询当前作用域的变量定义。
# globals()： 查询全局变量定义。



#
# def foo():
#     print(locals())
#     i = 1
#     # 申请一块内存，内存数据为 整数(1)
#     # 给一个变量标签(str)， "i"
#     # 变量标签(i) -> int(1)
#     # 关联关系 -> locals()
#     print(locals())
# foo()
```

课堂练习（5）-局部变量和全局变量

```
# 函数注册过程：
# 1、把代码块暂存放到内存中。
# 2、函数名称(str): "foo"
# 3、foo -> 暂存内存
# 4、关联关系 -> globals()：  注册

# print(globals())
# i = 1
# print(globals())
```

课堂练习（5）-局部变量和全局变量

```
# #全局变量
# value = 123
#
# def foo():
#     # 局部变量
#     # global value
#     value = 456
#     print(id(value))
#     print("foo:", value)
#     # print(locals())
#     print(globals())
#
# value = 789
# print(id(value))
# foo()
#
# cc = 1
# dd = 2
#
# foo()
#
# print("out:", value)
#
```

课堂练习（5）-局部变量和全局变量

```
# 传入复杂变量 与传入简单变量之间的处理上的不同。
# 请大家重点关注内存地址是否有变化。
# 传入简单变量时， 函数内修改，函数外不变。
# x = 10
# def foo(x):
#     x = 20
#     print(x,id(x))
# foo(100)
# print(x,id(x))
#
# 传入复杂变量时，若只修改修改元素，函数外同样会被修改。
# x = [10]
# def foo(x):
#     print(id(x))
#     x[0] = 20
#     x.append(200)
#     print(id(x))
#     print(x)
# foo(x)
# print(x,id(x))
```

课堂练习（2）- 全局变量与局部变量

- 写一个函数，验证 针对简单变量的全局变量 和局部变量区别。
- 写一个函数，验证针对 复杂变量的全局变量 和局部变量区别。

内置函数

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

内置参数详解https://docs.python.org/3/library/functions.html?highlight=built#ascii

递归函数

- 在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。
- 递归函数特性：
  - ✓ 必须有一个明确的结束条件；
  - ✓ 每次进入更深一层递归时，问题规模相比上次递归都应有所减少相邻两次重复之间有紧密的联系，前一次要为后一次做准备（通常前一次的输出就作为后一次的输入）。
  - ✓ 递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。
  - ✓ 由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出）

简单的例子：计算1到n之间相加之和；通过循环和递归两种方式实现

```python
def sum_calc(n):
    return sum(range(1,n+1))

print(sum_calc(10))
```

```python
def sum_calc(n):
    val = 0
    for i in range(1, n+1):
        val += i
    return val

print(sum_calc(10))
```

```python
def sum_calc(n):
    if n <= 1:
        return 1
    return n + sum_calc(n-1)

print(sum_calc(10))
```

课堂练习（4）- 递归函数

- 调用递归函数：
  - 1+2+ …. +n
  - 1*2*3*….*n

匿名函数 lambda

不需要显示给出名字的函数

- 代码更加简洁，使用lambda就可以省下定义函数过程，比如说我们只是需要写个简单的脚本来管理服务器时间，我们就不需要专门定义一个函数然后再写调用。
- 不为函数命名发愁，对于一些比较抽象并且整个程序执行下来只需要调用一两次的函数，有时候给函数起个名字也是比较头疼的问题，使用lambda就不需要考虑命名的问题了。
- 简化代码的可读性，阅读普通函数需要跳过开头def定义部分，使用lambda函数可以省去该步骤。

```
# func_dic = {"+": lambda x,y: x+y, "-":lambda x,y: x-y, "*":lambda x,y: x*y, "/":lambda x,y: x/y}
#
# def calc(x, y, flag):
#     func = func_dic.get(flag, None)
#     if func:
#         return func(x,y)
#     return None
#
# print(calc(1, 2, "+"))
```

课堂练习（4）- lambda

- 用lambda 去封装 +-*/ 算法。
  - 调用示例： calc(5.0, 1.0, “+”)
  - 需要考虑异常：
    - 输入变量 不是 整形数或浮点型数。
    - 针对除法，除数是0.0

课堂作业（5）

1. 编写一个函数：
    1. 生成1个给定长度n的自然数列表li = [1,…,n]
    2. 等待1秒。
    3. 返回1个元组，(函数用时,占用cpu时间,li)
2. 写函数，接收n个数字，求这些参数数字的和
3. 编写一个函数cacluate, 可以接收任意多个数,返回的是一个元组，元组的第一个值为所有参数的平均值, 第二个值是大于平均值的所有数.
4. 找出传入的列表或元组的奇数位对应的元素，并返回一个新的列表
5. 写一个函数，判断用户传入的列表长度是否大于2，如果大于2，只保留前两个，并将新内容返回给调用者
6. 写函数，统计字符串中有几个字母，几个数字，几个空格，几个其他字符，并返回结果。
7. 编写一个函数，利用递归函数，去实现2分查找功能。
8. 编写一个函数，实现如下功能：
    - calc(“+”, 10, 20), calc(“-”, 10, 20), calc(“*”, 10, 20), calc(“/”, 10, 20)

Python函数
进阶用法

本节重点：

- 高阶函数
- 嵌套函数与闭包
- 作用域与命名空间(LEGB)
- 语法糖-装饰器
- 语法糖-迭代器
- 异常处理

高阶函数

思考：传入的参数和返回结果能是函数对象吗？

- 答案：可以的！
- 变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。
- 例子： +-*/的使用。

```
# # 高阶函数示例
# def add(x,y):
#     return x+y
# def min(x,y):
#     return x-y
# def mul(x,y):
#     return x *y
# def div(x,y):
#     return x / y
# # 这是1个高阶函数，返回结果是函数
# def get_func(flag):
#     if flag == "+":
#         return add
#     if flag == "-":
#         return min
#     if flag == "*":
#         return mul
#     if flag == "/":
#         return div
#     return None
# # 这个也是一个高阶函数，传入参数有函数
# def do_calc(func, x, y):
#     return func(x,y)
# def calc(x, y, flag):
#     func = get_func(flag)
#     return do_calc(func, x, y)
# print(calc(1, 2, "+"))
```

```
# # 高阶函数示例
# def add(x,y):
#     return x+y
#
# def min(x,y):
#     return x-y
#
# def mul(x,y):
#     return x *y
#
# def div(x,y):
#     return x / y
#
# #
# # 善于使用 python提供的灵活强大的工具特性
# def get_func(flag):
#     func = globals().get(flag)
#     return func
# #
# # 这个也是一个高阶函数，传入参数有函数
# def do_calc(func, x, y):
#     return func(x,y)
#
# def calc(x, y, flag):
#     func = get_func(flag)
#     return do_calc(func, x, y)
#
# print(calc(1, 2, "add"))
# print(calc(1, 2, "min"))
# print(calc(1, 2, "mul"))
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - map: 生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后放入到新的数组中。
    - 举例：数组的平方，或取绝对值　　(1-1映射的关系)

```
# 数组的平方
#
# # 方法1：
# li = [1,2,3,4]
# nli = []
# for ele in li:
#     nli.append(ele * ele)
#
# # 方法2：
# nli = [ele * ele for ele in li]
```

```
# 方法3：
# def func(ele):
#     return ele * ele
# nli = map(func, li)
# # print(list(mp))
# for ele in nli:
#     print(ele)

# 方法4：
# def func(ele):
#     return ele * ele
# nli = map(lambda ele: ele * ele, li)
# # print(list(mp))
# for ele in nli:
#     print(ele)
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - filter:生成一个新数组，在遍历原数组的时候只将返回值为 true 的元素放入新数组。
    - 举例：过滤出所有的偶数/奇数。

```
# 过滤所有奇数
#
# nli = [ele for ele in li if ele % 2 > 0]
# print(nli)
#
# def func(ele):
#     return ele % 2 > 0

# nli = filter(lambda ele: ele % 2 > 0, li)
#
# print(list(nli))
```

高阶函数

- 3个典型应用（ map/filter/reduce ）：
  - reduce：将数组中的元素通过回调函数最终转换为一个值。
    - 举例：取最大值/最小值/平均值

```
# 求和
#
# from functools import reduce
# #
# # def func(e1,e2):
# #     return e1 + e2
#
# li = [1,10,100,1000]
# val = reduce(lambda e1, e2: e1 + e2, li)
# print(val)
```

```
# 求最大值
#
# from functools import reduce
#
# # def func(e1,e2):
# #     return e1 if e1 > e2 else e2
#
# li = [1,10,100,1000]
# val = reduce(lambda e1, e2: e1 if e1 > e2 else e2, li)
# print(val)
```

课堂练习（1）

1. 编写嵌套函数area()，专门计算图形的面积（兼容rect、circle、squre）
2. 利用map，实现1个函数，返回列表的平方数。
3. 利用map，实现1个函数，返回列表的绝对值。
4. 利用 filter，实现1个函数，返回列表的偶数项。
5. 利用reduce，实现1个函数，返回列表之和。
6. 利用reduce，实现1个函数，返回列表中元素的最大值
7. 人员信息字典：dic=[{‘name’:‘p1’, ‘salary’:200}, {‘name’:’p2’, “salary’:2000} , {‘name’:’p3’, “salary’:1500}]
   - 利用map，实现1个函数，所有人员的工资均增加 500元。
   - 利用 filter，实现1个函数，返回 工资大于 1000的人名列表
   - 利用reduce，实现1个函数，返回工资最高的人员名称
   - 利用reduce，实现1个函数，返回最高工资
   - 利用reduce，实现1个函数，返回平均工资

课堂练习（2）

写函数area()，专门计算图形的面积（兼容rect、circle、squre），按如下要求：

```
# # 方式1:
# def squre(x):
#     return x*x
#
# def rect(x, y):
#     return x * y
#
# def circle(r):
#     return 3.1415926 * r * r
#
# print(squre(10))
# print(rect(10, 20))
# print(circle(30))
#
```

```
# # 方式2:
# def area(tp, *args):
#     if tp == 'squre':
#         return squre(args[0])
#     if tp == 'rect':
#         return rect(args[0], args[1])
#     if tp == 'circle':
#         return circle(args[0])
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
#
```

```
# # 方式3:
# def area(tp, *args):
#     func = None
#     if tp == 'squre':
#         func = squre
#     if tp == 'rect':
#         func = rect
#     if tp == 'circle':
#         func = circle
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
#
#
# def circle2(x, y, z):
#     return x * y * z
#
```

```
# # 方式4:
# def area(tp, *args):
#     func = globals().get(tp, None)
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
# print(area('circle2', 30, 40, 50))
```

```
# # 方式5: 内嵌函数 + 高阶函数 + locals()
# def area(tp, *args):
#     def squre(x):
#         return x * x * 10
#
#     def rect(x, y):
#         return x * y * 10
#
#     def circle(r):
#         return 3.1415926 * r * r * 10
#
#     def circle2(x, y, z):
#         return x * y * z * 10
#
#     func = locals().get(tp, None)
#     if func == None:
#         print("没有找到处理函数")
#         return None
#     return func(*args)
#
# print(area('squre', 10))
# print(area('rect', 10, 20))
# print(area('circle', 30))
# print(area('circle2', 30, 40, 50))
```

课堂练习（3）

```
# map / filter / reduce
# 需求， 返回列表的平方数。
# li = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 方式1:
# li_new = []
# # for i in li:
# #     li_new.append(i * i)
# # print(li_new)

# for i in data:
#     pass

# 方式2: map(func, data) data: list/dict/set/str 。。。。
# def sqr(x):
#     return x * x

# 方式3: 匿名函数
# lambda x: x * x
```

课堂练习（4）

```
# # filter 的用法
# filter(func, data) -> 只有func 返回值  为True, data的数据才会插入新的列表
# print(list(filter(lambda a: True if a >= 0 else False, li)))
#
#
# def filter_func(x):
#     return x > 0
#
#
# print(list(filter(filter_func, li)))
# 1. 从 li中取出一个数; x
# 2. 调用 func()，  对数进行判断
# 3. 如果结果为True，则插入新的数组。

# filter 与 map的区别。
# map 不改变长度，但改变数值。
# filter 不改变数值，但改变长度。
```

课堂练习（5）

```
# reduce

# print(li)
# from functools import reduce
# 求和
# def reduct_func(x, y):
#     return x + y
# print(reduce(reduct_func, li))
# li=[5,2,3,4]
# 最小值
# def reduct_func(x, y):
#     print("x",x)
#     print("y",y)
#     return x if x < y else y
# print(reduce(reduct_func, li))
# 平均值
# def add(x, y):
#     return x + y
# print(reduce(add, li)/len(li))
```

课堂练习（6）

```
#
# oper_dict = {'+': lambda x, y: x+y,
#              '-': lambda x, y: x-y,
#              '*': lambda x, y: x*y,
#              '/': lambda x, y: x/y,
#              '**': lambda x, y: x **y}
# def calc(tp, *args):
#     func = oper_dict.get(tp, None)
#     if not func:
#         print("找不到。。。")
#         return None
#     return func(*args)
#
# print(calc('+', 2, 3))
# print(calc('-', 2, 3))
# print(calc('*', 2, 3))
# print(calc('/', 2, 3))
# print(calc('**', 2, 3))
```

嵌套函数与闭包

- 内嵌函数（允许在函数内部创建另一个函数，叫内部函数）
- 内部函数的整个作用域都在外部函数之内，如fun2整个定义和调用的过程都在fun1里面，除了在fun1里可以任意调用，出了fun1之外无法被调用，会系统报错。

嵌套函数示例1：

```
name = '1'
def fun1():
    name = '2'
    print('fun1 is using...', name)
    def fun2():
        name = '3'
        print('fun2 is using...', name)
    fun2()
    print('fun1 end...', name)
    name = '4'
fun1()
print('fun1 end...', name)
```

- 嵌套函数可以访问父函数中声明的所有局部变量、参数，嵌套函数在被外部调用时，就会形成闭包。
- 闭包的作用就是在父函数执行完并返回后，解释器垃圾回收机制不会收回父函数所占用的资源。
- 返回的函数对象，不仅仅是一个函数对象，在该函数外还包裹了一层作用域，该函数无论在何处调用，优先使用自己外层包裹的作用域

闭包示例2：

```
def funX(x):  # 外部作用域 的变量 x
    def funY(y):  # 内部函数（闭包）
        return x * y  # 引用了x变量

    return funY

fun = funX(8)
print(fun)  # <function funX.<locals>.funY at 0x000002889E0E8510>
print(type(fun))  # <class 'function'>
print(funX(8)(5))  # 40
print(fun(5))  # 40
```

命名空间（作用域）

又名name space, 存放变量标签与内存地址的绑定关系，python里面有很多名字空间，每个地方都有自己的名字空间，互不干扰，不同空间中的两个相同名字的变量之间没有任何联系。

名称空间有4种:LEGB

- locals:函数内部，包括函数局部变量以及形式参数
- enclosing function:在嵌套函数中外部函数的名字空间, 若fun2嵌套在fun1里，对fun2来说，fun1的名字空间就enclosing.
- globals: 当前的模块空间，模块就是一些py文件。也就是说，globals()类似全局变量。
- builtins: 内置模块空间，也就是内置变量或者内置函数的名字空间，print(dir(builtins))可查看。

不同变量的作用域不同就是由这个变量所在的名称空间决定的。
不同的空间，有不同的查询方式。

作用域查找顺序
当程序引用某个变量的名字时，就会从当前名字空间开始搜索。搜索顺序规则便是:LEGB。即locals -> enclosing function -> globals ->builtins。一层一层的查找，找到了之后，便停止搜索，如果最后没有找到,则抛出在NameError的异常。

作用域示例（LEGB）

```python
# LEGB local - enclosed - global - buildin

# name = 'global_name'  #  全局
# def func():
#     name = 'enclosed_name'  # 闭包
#     # name2 = 'cc'
#     def func1(): # 内部函数
#         # name = 'local_name'  # 本地
#         return name
#     return func1
#
# a = func() # 高阶函数
# print(a())
#
# print(globals())

#print(a)
# name = 'global_name2'

# print(a())
```

```python
# # python 解析机制，以及 变量/函数的注册流程
# def oper(name):
#     #1.  'name':name -> locals()
#     def add(x, y):
#         z = x + y
#         print(name, locals())
#         return z
#
#     #2. 'calc': calc -> locals()
#     def mul(x, y):
#         z = x * y
#         print(name, locals())
#         return z
#     #3. 'calc': calc -> locals()
#     print("oper::locals()", locals())
#
#     func = locals().get(name, None)
#     if not func:
#         print("error", name)
#     return func
#     # if name == 'add':
#     #     return add
#     # if name == 'mul':
#     #     return mul
#
# # func = oper("mul")
# # val = func(2, 3)
#
# val = oper("add")(2,3)
# print(val)
```

```python
# python 解析的规则
# = / += / -= / *= / /=,
# 符号左边的变量会被python认为是局部变量，除非global 来提前指定。
#
# a = 1
# def foo():
#     def bar():
#         a += 1
#         return a
#     return bar
#
# c = foo()
# val = c()
# print('第1次:', val)
# val = c()
# print('第1次:', val)
```

作用域示例

```
def func():
    # name = 'enclosed_name'
    def sub_func():
        # name = 'local_name'
        return name
    return sub_func
name = "global_name"
a = func()
b = a()
print(b)
```

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
c =foo()
c()
```

- 这段程序的本意是要通过在每次调用闭包函数时都对变量a进行递增的操作。但在实际使用时
- 在执行代码 c = foo()时，python会导入全部的闭包函数体bar()来分析其的局部变量，python规则指定所有在赋值语句左面的变量都是局部变量，则在闭包bar()中，变量a在赋值符号"="的左面，被python认为是bar()中的局部变量。再接下来执行print c()时，程序运行至a = a + 1时，因为先前已经把a归为bar()中的局部变量，所以python会在bar()中去找在赋值语句右面的a的值，结果找不到，就会报错

```
flist = []
for i in range(4):
    print(id(i))
    def foo(x):
        print(id(i))
        print(x + i)
    flist.append(foo)

print("this:", id(i))
i = 2
for f in flist:
    f(2)
```

```
flist = []
for i in range(4):
    print(id(i))
    def foo(x, y=i):
        print(x + y)
    flist.append(foo)

print("this:", id(i))
i = 2
for f in flist:
    f(2)
```

分析一下计算结果，思考为什么？

作用域示例

```
# LEGB 变量引用顺序
# L: LOCAL
# E：ENCLOSED
# G：GLOBALS
# B: buildins
# name = 'global_name'  # 1
# def func():
#     name = 'enclosed_name'  # 2
#     # name2 = 'cc'
#     def func1():
#         name = 'local_name'  # 3
#         # print(name2)
#         return name
#     return func1
#
# a = func()
# print(a())
# print(a)
# name = 'global_name2'
#
# print(a())
```

课堂练习（6）

用python 的闭包机制， 去实现 文件检索功能，示例：

- func_1 = get_file_info(“file.txt”)
- count_1 = func_1(“中国”) # 得到文件中的“中国”字符串的数量。
- count_1 = func_1(“美国”) # 得到文件中的“美国”字符串的数量。

```
def get_file_info(file_name):
    li = []
    with open(file_name, "r", encoding='utf-8')  as f:
        li = f.readlines()
    def inter(name):
        num = 0
        for str in li:
            num += str.count(name)
        print(name, num)
        return num
    return inter

func = get_file_info("file.txt")
func("中国")
func("美国")
```

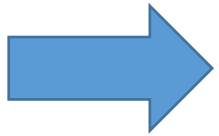
语法糖-装饰器-不推荐用法1

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式1

```
def login():
    print("come to login...")


def func1():
    login()
    print("come to func1...")


func1()
func2()
func3()
```

每个模块都需要修改，直接违反了软件开发中的一个原则“开放-封闭”原则，

简单来说，它规定已经实现的功能代码不应该被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块不应该被修改
- 开放：对现有功能的扩展开放

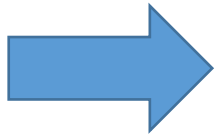
语法糖-装饰器-不推荐用法2

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式2

```
def login():
    print("come to login...")

def func1():
    print("come to func1...")

def login_func1():
    login()
    func1()
    print("come to func1...")

login_func1()
login_func2()
login_func3()
login_func4()
```

模块本身没有动，但需要修改调用该模块的方式（func1 -> login_func1）。同样违反了软件开发中的一个原则“开放-封闭”原则。

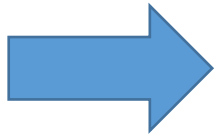
语法糖-装饰器-常规用法

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
需要增加登录功能



方式3

```
def login():
    print("come to login...")

def func1_nologin():
    print("come to func1...")

def func1():
    login()
    func1_nologin()

func1()
func2()
func3()
func4()
```

工程上，可考虑使用的方式，c/c++可考虑此方法

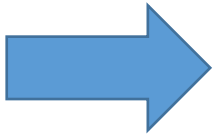
语法糖-装饰器-python推荐用法

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

原始代码

```
def func1():
    print("come to func1...")

def func2():
    print("come to func1...")

def func3():
    print("come to func1...")

def func4():
    print("come to func1...")

func1()
func2()
func3()
func4()
```

需求有变化：
增加登录功能



方式4

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

def func1():
    print("come to func1...")

func1 = login(func1)

def func2():
    print("come to func2...")

func2 = login(func2)

func1()
func2()
```

方式5

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

@login
def func1():
    print("come to func1...")

@login
def func2():
    print("come to func2...")

func1()
func2()
```

课堂练习（1）- 打印时间

```
# 示例1：
# 给调用的函数 添加 时间戳语句 -> 进入时打印时间，函数执行完成时，也打印时间。

# import time
#
# def timer(func):
#     def inner():
#         print("come to func ... ", time.time())
#         func()
#         print("end func ... ", time.time())
#     return inner
#
# @timer
# def func1():
#     print("func1....")
#
# @timer
# def func2():
#     print("func2....")
#
#
# func1()  -> logger(func1)
# func2()  -> logger(func2)
```

课堂练习（2）-给函数添加运行日志 用装饰器来修饰带参数的函数

```
# 示例2：  给函数添加运行日志 用装饰器来修饰带参数的函数.
# 这里需要使用到了不定参数 *args, **kwargs
# def logger(func):
#     def inner(*args, **kwargs):
#         print("come to func ...")
#         ret = func(*args, **kwargs)
#         print("end func ...")
#         return ret
#     return inner
#
# @logger
# def squre(x):
#     return x * x
#
# @logger
# def rect(x,y):
#     return x * y
#
# @logger
# def circle(r):
#     return 3.1415926 * r * r
#
# print(squre(10)) # -> login(squre)(10) -> inner(10)
# print(rect(10, 20))
# print(circle(10))
#
```

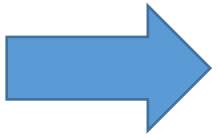
语法糖-装饰器-带参数运行

任务需求：做一个网站，有4个功能， func1, func2, func3, func4

方式5

```
def login(func):
    def inter():
        print("come to login...")
        func()

    return inter

@login
def func1():
    print("come to func1...")

@login
def func2():
    print("come to func2...")

func1()
func2()
```

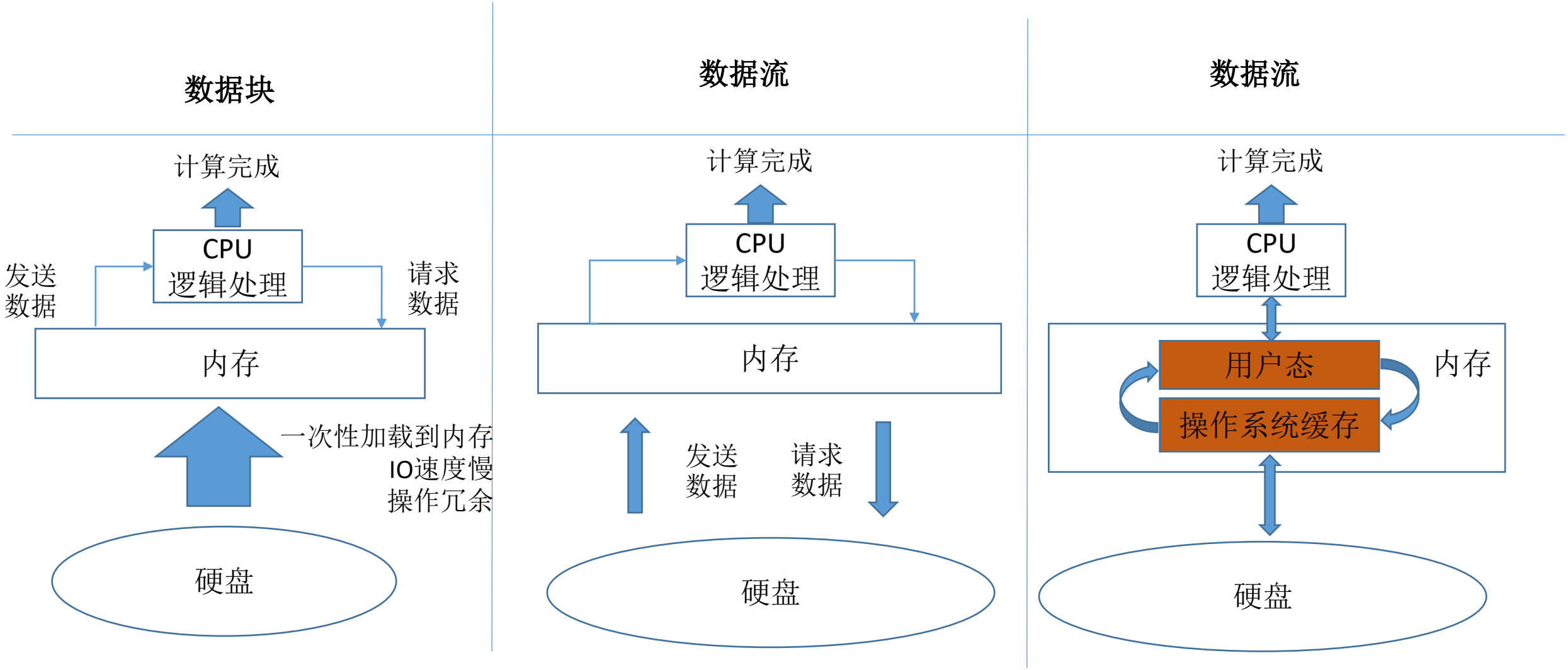
带参数运行



```
def login(func):
    def inter():
        print("come to login...")
        func()
    return inter

@login
def func1(para1):
    print("come to func1...")

@login
def func2(para1, para2):
    print("come to func2...")

@login
def func3(para1, para2, para3):
    print("come to func3...")

@login
def func4(para1, para2, para3,
para4):
    print("come to func4...")

func1(1)
func2(1,2)
func3(1,2,3)
func4(1,2,3,4)
```

```
def login(func):
    def inter(*args, **kwargs):
        print("come to login...")
        func(*args, **kwargs)
    return inter

@login
def func1(para1):
    print("come to func1...", para1)

等价于： login(func1)(para1)

@login
def func2(para1, para2):
    print("come to func2...", para1, para2)

@login
def func3(para1, para2, para3):
    print("come to func3...", para1, para2, para3)

@login
def func4(para1, para2, para3, para4):
    print("come to func4...", para1, para2, para3,
para4)

func1(1)
func2(1,2)
func3(1,2,3)
func4(1,2,3,4)
```

语法糖-生成器


数据块
计算完成
CPU
逻辑处理
发送
数据
请求
数据
内存
一次性加载到内存
IO速度慢
操作冗余
硬盘
数据流
计算完成
CPU
逻辑处理
内存
发送
数据
请求
数据
硬盘
数据流
计算完成
CPU
逻辑处理
用户态
操作系统缓存
内存
硬盘

语法糖-生成器

数据块	数据流
把数据看成数据块，一次性读入到内存，直接使用， 解析和导入时间长，内存压力大 大文件不适用	把硬盘上的数据看成流，一边读入，一边处理，直到逻辑结束
离线视频文件，必须全部下载完成才能看	在线视频，边下载边看。
提前生成数据，用于后续调用	通过generator来生成，用时再生成并调用

语法糖-生成器generator

```
for i in range(100000000):
    if i == 50:
        break
    print(i)
```

问题：

创建一个包含10000万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

怎么办？

```
xr = (x * x for x in range(10))
print(type(xr))
print(xr)
for x in xr:
    print(x)
```

问题：

生成器能否转化成 list or tuple or set?

注意：

python3中，range()已经做了优化

语法糖-函数生成器 yield

```
def show_vlaue():
    for i in range(100):
        val = yield i
        print(val)

a = show_vlaue()
for i in a:
    if i == 10:
        a.send(10)
    print(i)
```

如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator。

这里，最难理解的就是generator和函数的执行流程不一样。

函数是顺序执行，遇到return语句或者最后一行函数语句就返回

而变成generator的函数，在每次调用next()的时候执行，遇到yield语句暂停并返回数据到函数外，再次被next()调用时从上次返回的yield语句处继续执行。

注意：调用send(x)给生成器传值时，必须确保生成器已经执行过一次next()调用，这样会让程序走到yield位置等待外部第2次调用。

语法糖-函数生成器 yield

作用1：分块读取文件

```
def read_file_block(file_name):
    block_size = 1024
    with open(file_name, 'rb') as f:
        while True:
            block = f.read(block_size)
            if not block:
                return
            yield block

f = read_file_block("top.png")
i = 1
for data in f:
    print(i)
    i += 1
    print(data)
```

练习，看得到什么结果？

```
def h():
    print('hello')
    x = yield 6
    print(x)
    y = yield 23
    print(y)
c = h()
x = c.__next__()
#x 获取了yield 6 的参数值 6
y = c.send('ok')
#y 获取了yield 23 的参数值23
print(x,y)
```

语法糖-函数生成器的作用

需求：返回素数的生成器

```
# 下面是 is_prime 的一种实现...
def is_prime(number):
    if number <= 1:
        return False
    if number == 2:
        return True
    if number % 2 == 0:
        return False
    for current in range(3, int(number / 2 + 1), 2):
        if number % current == 0:
            return False
    return True

def get_primes(value):
    return (element for element in range(2, value + 1)
if is_prime(element))

b = get_primes(10000000000000)
for i in b:
    if i > 100:
        break
    print(i)
```

降低计算量

```
def get_primes(value):
    all_primes = []
    for i in range(2, value):
        is_prime = True
        for k in all_primes:
            if i % k == 0:
                is_prime = False
                break
        if is_prime:
            all_primes.append(i)
            yield i

n_count = 0
b = get_primes(10000000000000)
for i in b:
    if i > 100000:
        break
    n_count += 1
    print(n_count, i)
```

思考：all_primes，能否变成全局变量？

当一个生成器函数调用yield，生成器函数的“状态”会被冻结，所有的变量的值会被保留下来，下一行要执行的代码的位置也会被记录，直到再次调用next()。一旦next()再次被调用，生成器函数会从它上次离开的地方开始。如果永远不调用next()，yield保存的状态就被无视了。

语法糖-可迭代对象 and 迭代器 and 生成器

- 可以直接作用于for循环的对象统称为可迭代对象（Iterable，可遍历、可循环），可以使用isinstance()判断一个对象是否是iterable对象，主要分为两大类：
  - 集合数据类型，如list、tuple、dict、set、str等；
  - generator，包括生成器和带yield的generator function。
- 被next()函数调用并不断返回下一个值的对象称为迭代器：iterator，所有生成器都是Iterator对象。
- list、dict、str虽然是iterable，却不是iterator，但可以通过iter()函数变成iterator。
- 为啥要引入iterator对象？
  - 通过iterator，python就可以处理长度不定（或无限大）的有序序列数据流，处理中，通过不断通过next()函数实现按需计算下一个数据，直到抛出StopIteration异常。
  - Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的（）。
  - Iterator可以处理硬盘数据的读取（长度未知）。
  - Iterator可以读取socket传送的网络报文数据流（长度未知）。

```
x = iter(range(1, 100))

for i in x:
    print(i)

while True:
    try:
        i = next(x)
        print(i)
    except StopIteration:
        break
```

课堂练习（2）

1. 利用装饰器， 添加函数的开始时间，和结束时间
2. 利用装饰器， 实现函数调用前的登录验证（只需要登录1次）。
3. 利用函数生成器， 编写函数，得到素数的生成器。
4. 利用函数生成器， 编写函数，监测某文件中是否有某字符串。

```python
def get_file_info(file_name):
    with open(file_name, "r", encoding='utf8') as f:
        for ln in f:
            yield ln

def jude_str_in_file(file_name, find_str):
    file_inter = get_file_info(file_name)
    for ln in file_inter:
    if ln.count(find_str):
        return True
    return False

file_name = "user.txt"
find_str = "abc"

flag = jude_str_in_file(file_name, find_str)
print(flag)
```

课堂练习（3）-登录验证

```
# # 示例3， 登录验证
# login_status = False
# def login(func):
#     def inner(*args, **kwargs):
#         global login_status
#         if not login_status:
#             print("come to login...")
#             login_status = True
#         print("come to func...")
#         ret = func(*args, **kwargs)
#         print("end func ...")
#         return ret
#     return inner
#
# @login
# def func1():
#     print("func1...")
#
# @login
# def func2():
#     print("func2...")
#
#
# func1()
# func2()
```

异常处理

异常就是程序运行时发生错误（包括语法错误、逻辑错误）的信号，若程序产生了异常信号，且没有处理它，则会抛出该异常，程序的运行也随之终止。常见的错误异常类型：

- IndentationError 语法错误（的子类）；代码没有正确对齐
- SyntaxError Python代码非法，代码不能编译
- AttributeError 试图访问一个对象没有的树形，比如foo.x，但是foo没有属性x
- IOError 输入/输出异常；基本上是无法打开文件
- ImportError 无法引入模块或包；基本上是路径问题或名称错误
- IndexError 下标索引超出序列边界，比如当x只有三个元素，却试图访问x[5]
- KeyError 试图访问字典里不存在的键
- KeyboardInterrupt Ctrl+C被按下
- NameError 使用一个还未被赋予对象的变量
- TypeError 传入对象类型与要求的不符合
- ValueError 传入一个调用者不期望的值，即使值的类型是正确的

异常处理：几种典型用法

```
try:
    raise TypeError('类型错误')
except Exception as e:
    print(e)
```

```
s1 = 'hello'
try:
    int(s1)
    print( 'a' )
except IndexError as e:
    print(e)
except KeyError as e:
    print(e)
except ValueError as e:
    print(e)
except Exception as e:
    print(e)
```

```
s1 = 'hello'
try:
    int(s1)
except IndexError as e:
    print(e)
except KeyError as e:
    print(e)
except ValueError as e:
    print(e)
#except Exception as e:
#    print(e)
else:
    print( 'try内代码块没有异常则执行我' )
finally:
    print('无论异常与否,都会执行该我')
```

```
class EgonException(BaseException):
    def __init__(self, msg):
        self.msg = msg
    def __str__(self):
        return self.msg
try:
    raise EgonException('类型错误')
except EgonException as e:
    print(e)
```

- 把错误处理和真正的工作分开来
- 代码更易组织，更清晰，复杂的工作任务更容易实现；
- 毫无疑问，更安全了，不至于由于一些小的疏忽而使程序意外崩溃了；

课堂练习（5）-异常处理-实验1

```
# i = 1
# a = []
# b = []
# try:
#     while True:
#         j = i + 1
#         i += 1
#         # a = int('aaa')
#         if i >= 10:
#             break
#         raise ValueError("haaha")
# except NameError as e:
#     print(e)
# except IndentationError as e:
#     print(e)
# except ValueError as e:
#     print(e)
# else:
#     print("没有异常！")
# finally:
#     print("有没有异常都会执行到我！")
# print(i)
#
```

课堂练习（3）- 异常练习

- 用异常来重新封装一下文件copy 程序。
- 编写函数, 接收一个列表(包含30个1~100之间的随机整形数)和一个整形数k, 返回一个新列表.

  函数需求:

  将列表下标k之前对应(不包含k)的元素逆序;

  将下标k及之后的元素逆序;

  例如：

  输入两个参数：[1,2,3,4,5] 2， 返回结果： [2,1,5,4,3]

  注意：异常处理：

  （1）输入非列表

  （2）k不在范围之内

课堂练习（6）-异常处理-实验2-文件测试

```
# def read_file(file_name):
#     try:
#         f = open(file_name, 'rb')
#         cc = f.read()
#         print(cc)
#         f.close()
#     except FileNotFoundError as e:
#         # add your err del code!
#         print(e)
#
# read_file('abb.txt')


# def read_file(file_name):
#     with open(file_name, 'rb') as f:
#         print(f)
#         cc = f.read()
#         print(cc)
#
# read_file('abb.txt')
```