函数：一种编程思维

函数初识

定义：def--关键字

将某个功能封装到一个空间中就是一个函数

功能：

减少重复代码

函数的调用

函数名+（）：调用函数和接收返回值

函数的返回值

1. return 值 == 返回值
2. 可以返回任意的数据类型
3. return 返回多个内容是元祖的形式
4. return 下方不执行，并且会终止当前函数
5. return 不写或者写了return 后面不写值都会返回None
6. 函数的返回值返回给函数的调用者
7. 函数的返回值可以有多个结果

函数的参数

1. 分类：
    1. 位置参数：一一对应
    2. 默认参数：参数定义是括号中写好的就是默认参数（不进行传参就使用默认参数，使用传参时使用传递的参数）
    3. 关键字参数，按照关键字进行传参
    4. 位置参数必须放在默认参数之前
    5. 混合参数：位置参数和关键字参数一起传参
2. 形式参数：函数定义阶段括号中的参数叫做形参
3. 实参：函数调用阶段括号中的参数叫做是实参
4. 传参：将实参传递给形参的过程叫传参

三元运算：

格式：条件成立的结果　条件　条件不成立的结果

参数：

*args，**kwargs：

1. *args（函数定义阶段代表聚合，返回的是一个元祖）： 大家共用的一个名字，可以修改但是不建议
2. **kwargs（聚合关键字参数，返回的是一个字典）

```
def eat(*args): #函数的定义阶段     *聚合（打包）
    print(args) #元祖
    print(*args)#函数体中的*，打散（解包）
```

优先级

- 位置参数 > 动态位置参数 > 默认参数 > 动态默认参数

函数体中的*代表什么

- 第一个代表：聚合
- 第二个代表：打散

形参中参数定义的位置：

- 位置参数：定义在函数体开头的时候
- 动态位置参数：先执行位置参数，位置参数接收后额外的参数动态位置进行接收，获取到的是一个元祖
- 默认参数：函数接受体接收到的参数
- 动态关键字参数：先执行默认参数，默认参数接收后，额外的默认参数动态进行接收，获取到的是一个字典

实参中的位置：

- “*” 打散
- “**” 实参可以使用

函数的注释

1. 查看注释：.doc
2. 查看函数的名字：.name

名称空间

内置空间

- python解释器自带的一块空间

全局空间

- py文件中顶格写就是全局空间

局部空间

- 函数体中就是局部空间

加载顺序：

内置空间 > 全局空间 > 局部空间

取值顺序

局部空间 > 全局空间 > 内置空间

作用域

1. 全局作用域：全局空间+内置空间
2. 局部作用域：局部空间

函数的嵌套

- 不管在什么位置，函数名+括号 （）就是在调用一个函数

修改全局：

global：修改全局变量

nonlocal：修改局部，修改离他最近的一层，上一层没有继续向上层查找，只限局部

函数名的使用及第一类对象

1. 函数名可以当做值赋值给变量
2. 函数名可以当做容器中的元素
3. 函数名可以当做函数的参数
4. 函数名可以当做函数的返回值

f-strings

f"{变量名}"

F"{变量名}"

f"""{变量名}"""

迭代器：是基于上一次停留的位置，进行取值

可迭代对象

1. 定义：只要具有 __ iter __ 方法的就是一个可迭代对象
2. list，tuple，str，set，dict取值方式只能直接看、

迭代器：具有 __ iter __ () 和 __ next __ () 两个方法的才是迭代器

```
lst = [1,2,3,4,5]
 l = lst.__iter__()  # 将可迭代对象转换成迭代器

 l.__iter__()  # 迭代器指定__iter__()还是原来的迭代器
 print(l.__next__())   # 1
 print(l.__next__())   # 2
```

for循环的本质：

```
while True:
    try:
        print(l.__next__())
    except StopIteration:
        break
```

优点：

- 惰性机制，可以节省内存空间

缺点：

1. 不能直接查看值，迭代器查看到的是一个迭代器的内存地址

2. 一次性，用完就没有了
3. 不能逆行，后退，只能继续执行下一条

空间换时间：

- 容器存储大量数据，取值快，占用空间大

时间换空间：

- 迭代器就是一个大容器，节省了空间，但是取值慢

迭代器中Python 2 和 Python 3的区别、

1. python 3
    - iter （）和 __ iter __() 都有
    - next （） 和 __ next __ () 都有
2. Python 2
    - iter （）和 __ iter __()
    - next （）

生成器：本质就是一个迭代器

区别：

1. 迭代器是Python 自带的
2. 生成器是程序员自己写的一种迭代器

编写方式：

1. 基于函数编写
2. 推导式编写

yield

- 内存地址函数体中出现yield代表要声明一个生成器，
- generator -- 生成器 获取到的是一个生成器的内存地址
- 一个yield必须对应一个next

用途：节省空间（买包子的例子）

使用场景：

- 当文件或容器中数据量较大时，建议使用生成器

可迭代对象，迭代器，生成器比较：

1. 可迭代对象：（python 3 ）
    - 优点：list，tuple，str 节省时间，取值方便，使用灵活（具有自己的私有办法）
    - 缺点：消耗内存
2. 迭代器：
    - 优点：节省空间
    - 缺点：不能直接查看值，使用不灵活，消耗时间，一次性，不可逆
3. 生成器：
    - 优点：节省空间，人为定义
    - 缺点：不能直接查看值，消耗时间，一次性，不可逆行

yield 和return的区别

1. 相同点：
    1. 都是返回内容
    2. 都可以返回多次，但是return写多个只会执行一个
2. 不同点：
    1. return会终止函数，yield 是暂停生成器
    2. yield能够记录当前执行位置

迭代器和生成器的区别

- 通过send方法区别迭代器和生成器
- 通过内存地址区分

```
# 数据类型 (pyhton3: range() | python2 :xrange()) 都是可迭代对象 __iter__()
# 文件句柄是迭代器  __iter__() __next__()
```

yield from 和yield的区别：

yield from 将可迭代对象逐个返回

yield 将可迭代对象一次性返回

推导式

列表推导式：

1. 循环模式：（[变量 for循环]）
    - print([i for i in range(10)])
2. 筛选模式：[加工后的变量 for循环 加工条件]
    - print （[i for i in range（10） if i > 2]）
    - print （[i for i in range（10） if i % 2 == 0]）

集合推导式

1. 普通循环：{变量 for循环}
    - print （ {i for i in range（10）} ）
2. 筛选模式：{加工后的变量for循环 加工条件}
    - print({i for i in range (10) if i % 2 == 1})

字典推导式：

1. 普通：{键：值 for循环}
    - print {"key":"value" for i in range (10)}
2. 筛选：{加工后的键：值 for循环 加工条件}
    - print({i : i+1 for i in range(10) if i % 2 == 0})

生成器推导式（面试）：

1. 普通模式：（变量 for循环）

- tu = （i for i in range(10)）这是一个生成器

2. 筛选模式：（加工后的变量 for循环 加工条件）

```
tu = （i for i in range（10）if  i > 5）
    for i in tu :
        print(i)
```

内置函数：

1. eval：执行字符串类型的代码
2. exac：执行字符串类型的代码
3. hash（）：区分可变数据类型和不可变数据类型
4. help（）：查看帮助信息
5. callable（）：查看对象是否可以调用
6. int（）：将字符串或数字转换成整型
7. float（）：转换成浮点数
8. complex（）：复数
9. bin（）：十进制转换成二进制
10. oct（）：十进制转换成八进制
11. hex（）：十进制转换成十六机制
12. divmod（）：计算除数与被除数的结果，包含一个商和余数的元祖
13. round（）：保留浮点数的小数位数，可以设定保留位数，默认保留整数
14. pow（）：求X ** y次幂（三个参数的时候为x**y的结果对第三个参数取余）
15. bytes（）：用于不同编码之间的转换，建议使用encode
16. ord（）：通过元素获取当前表位的编码位置
17. chr（）：通过表位序号查找对应的元素
18. repr（）：查看数据的原生态（给程序员使用的）
19. all（）：判断容器汇总的元素是否都为真
20. any（）：判断容器中的元素有一个为真
21. 两个字典合并成一个：
    1. update--dic2.updata(dic1)
    2. 打散：** dic1 ,**dic 2
    3. print(dict([(1,2),(3,3)]))---括号里面的数字多一个少一个都不行，列表进行迭代，元祖也可以
    4. dict(** dict1 ,** dict2)
22. set():将可迭代对象转换成元祖
23. sep（）：每一个元素之间的分割方法

```
print(1,2,3,sep = "|")
输出结果：1|2|3
```

24. end：print执行完后的结束语句，默认\n

```
print(1,2,3,end = "")
print()
输出结果：全部在一行
```

25. print（）：屏幕输出
26. file：文件句柄，默认显示到屏幕
27. sum() -- 求和，必须是可迭代对象，对象中的元素必须为整型，字符串类型不能使用

```
print (sum ([1,2,3,1]))
print (sum ([1,2,3,1],100)      #100是起始值，就是从100开始进行取和，指定开始位置的值
```

28. abs() : 返回绝对值--转换成正数，不管你是不是负数
29. dir() : 查看当前对象具有什么方法
30. zip()：拉链，当可迭代对象的长度不一致时，选择最短的进行合并，可以是多个

```
面试题：
list1 =[1,2,3,4]
lst2 = ["alex","wusir","meet"]
print(list(zip(lst1,lst2)))

输出结果：[(1,"alex"),(2,"wusir"),(3,"meet")]

print(dict(zip(lst1,lst2))) #返回一个字典
输出结果：{1:"alex",2:"wusir",3:"meet"}
```

31. format(): 格式转换
    1. 对齐方式

```
print(format("alex",">20")) #右对齐
print(format("alex","<20")) #左对齐
print(format("alex","^20")) #居中
```

    1. 进制转换

```
print(format(10,"b"))       #十进制转二进制
print(format(10,"08b"))     #不够就补0
```

```
print (format(10,"08o"))        #oct,八进制
print (format(10,"08x"))        #hex，十六进制
```

```
print(format(0b1010,"d"))   #二进制转十进制
```

32. reversed() : 将一个序列进行翻转，返回翻转序列的迭代器

```
l = reversed('你好')  # l 获取到的是一个生成器
print(list(l))
ret = reversed([1, 4, 3, 7, 9])
print(list(ret))  # [9, 7, 3, 4, 1]
```

匿名函数：lambda

1. 函数名 = lambda 参数： 返回值
2. 匿名函数的名字叫做lambda
3. lambda是定义函数的关键字，相当于函数的def
4. 只可以返回一个数据类型，
5. lambda == def == 关键字
    1. lambfa x: x
        - x:是普通函数的形参（位置，默认），可以写任意多个，也可以不写
        - ： 后边是普通函数的返回值,必须写，没有默认，必须要有返回值，只能写一个数据类型

```
面试题：
print ([lambda : i for i in range(5)])
（返回5个内存地址）

print(lst[0]())---结果是4--因为循环最后一次输出是4，调用的全局最后一个就是4
不加后面的小括号就是调用内存地址
加了括号就是调用函数
```

```
lst = [lambda x : x+1 for i in range(5)]
print(lst[0](5))
输出结果为：6，返回值是x+1
```

```
tu = (lambda : i for i in range(3))
print(tu[0])    #输出错误，不能索引
print（tu）#输出内存地址
print(next(tu))     #一个函数地址
print(next(tu)())       #输出0
print(next(tu)())       #输出1
```

```
  ```python
  lst = [lambda : i for i in range(3)]
  print(lst[0]())
  tu = (lambda : i for i in range(3))
  print(next(tu)())
  输出结果：
  2
  0
  ```

  函数体中存放的是代码

  生成器中存放的也是代码

        --原因：yield导致函数和生成器的执行结果不一致
```

```python
lst = [lambd x:x+5  for i in range(2)]
print([i(2) for i in lst])


解开顺序：
lst = []
for i in range (2):
    lst.append(lambda x :x+5)
new_list
for i in lst:
    print(i)       #两个函数的内存地址
    new_list.append(i(2))
print(new_list)        #输出列表【7，7】

输出结果：【7，7】



lst = (lambda x:x+5  for i in range(2))
print([i(2) for i in lst])



输出结果：【7，7】

lst = （lambda x:x*i  for i in range(2)）
print([i(2) for i in lst])        #【0，2】


```
```

高阶函数

1. filter：筛选过滤

```
语法：filter(function,iterable)
function：1.指定过滤规则（函数的内存地址） 2.用来筛选的函数,在filter中会自动的把iterable中的元素传递给function,然后根据function返回的True或者False来判断是否保留此项数据

iterable:可迭代对象

####写函数名切记不加括号

lst = [{'id':1,'name':'alex','age':18},
       {'id':1,'name':'wusir','age':17},
       {'id':1,'name':'taibai','age':16},]

ls = filter(lambda e:e['age'] > 16,lst)

print(list(ls))

结果：
[{'id': 1, 'name': 'alex', 'age': 18},
 {'id': 1, 'name': 'wusir', 'age': 17}]
```
```

2.map（）：映射函数--将每个元素都执行了指定的方法

```
print(list(map()))-- 格式

语法: map(function,iterable) 可以对可迭代对象中的每一个元素进映射,分别取执行function，计算列表中每个元素的平方,返回新列表

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

print(list(map(lambda s:s*s,lst)))

计算两个列表中相同位置的数据的和

lst1 = [1, 2, 3, 4, 5]

lst2 = [2, 4, 6, 8, 10]

print(list(map(lambda x, y: x+y, lst1, lst2)))

结果:

[3, 6, 9, 12, 15]
```

3. sorted（）：排序函数，默认返回的是列表

```
语                                法:sorted(iterable,key=None,reverse=False)

print(sorted([1,-22,3,4,5,6],key = abs))          #key指定排序规则，abs后不加括号


在sorted内部会将可迭代对象中的每一个元素传递给这个函数的参数.根据函数运算的结果进行排序

lst = [{'id':1,'name':'alex','age':18},
    {'id':2,'name':'wusir','age':17},
    {'id':3,'name':'taibai','age':16},]

# 按照年龄对学生信息进行排序
print(sorted(lst,key=lambda x : list(x.values()))  #  值排序
print(sorted(lst,key=lambda e:e['age']))

结果:
[{'id': 3, 'name': 'taibai', 'age': 16}, {'id': 2, 'name': 'wusir', 'age':
17}, {'id': 1, 'name': 'alex', 'age': 18}]
```

4. max（）：最大值

```
print(max(10,3,4,5,2,6,76))      #输出  76
print(max(10,3,4,5,2,-6,76), key= abs)
```

5. min（）：最小值，可迭代对象，key=指定规则

6. reduce() :--计算，一层一层垒起来算，指定的函数方法必须接受两个形参

```
print(reduce(lambda x,y:x+y,[1,2,3,4,5]))
```

闭包：

1. 定义：在嵌套函数体内，使用非本层变量和非全局变量的罪就是闭包
    1. 闭包必须是内层函数对外层函数的变量（非全局变量）的引用
    2. 函数体执行完毕后，函数体内的空间自行销毁

```
def func():
    a=1
    def foo():
        print(a)
    print(foo.__closure__)  #判断是不是闭包，返回None就不是闭包
func()
```

2. 作用：
    1. 保护数据的安全性
    2. 装饰器

装饰器

原则：开放封闭原则

- 在不修改源代码及调用方式的前提下，对功能进行额外添加就是开放封闭原则

1. 开放：对代码的扩展进行开发
2. 封闭:修改源代码

标准版装饰器

```
# print(time.time())    #时间戳，小数


import time
def func():
       time.sleep(1)
       print("这是小刚写的功能")

def func():
       time.sleep(1)
       print("这是小红写的功能")

def index():
    time.sleep(2)
    print("这是小明写的功能")


def times(func):    #func == index 函数内存地址
    def foo():
        start_time = time.time()    #时间戳，被装饰函数执行前干的事
        func()  #fun == index ，后面加括号就是调用函数
        print(time.time()-start_time)#被装饰函数执行后干的事
        return foo

index = time(index)          #index== foo内存地址
index()     #index() == foo ()
func = time(func)
```

```
func()
```

语法糖：要将语法糖放在被装饰的函数上方

```
# def warpper(f):
#     def inner():
#         print("111")
#         f()
#         print("222")
#     return inner
#
# @warpper  # func = warpper(func)
# def func():
#     print("被装饰的函数1")
#
# @warpper  # index = warpper(index)
# def index():
#     print("被装饰的函数2")
#
# func()
# index()

# def warpper(f):
#     def inner(*args,**kwargs):
#         print("被装饰函数执行前")
#         ret = f(*args,**kwargs)
#         print("被装饰函数执行后")
#         return ret
#     return inner
#
# @warpper
# def func(*args,**kwargs):
#     print(f"被装饰的{args,kwargs}")
#     return "我是func函数"
#
# @warpper
# def index(*args,**kwargs):
#     print(11111)
#
# print(func(1,2,3,4,5,6,7,8,a=1))
```

装饰器的进阶

有参装饰器：

```
def warpper(func):
    def inner(*args,**kwargs):
        user = input("user:")
        pwd = input("pwd:")
        if user == 'alex' and pwd == "dsb":
            func(*args,**kwargs)
    return inner

@warpper
def foo():
    print("被装饰的函数")

foo()
```

面试题:

```
多个装饰器装饰一个函数
多个装饰器装饰一个函数时,先执行离被装饰函数最近的装饰器
```

```
def auth(func): # wrapper1装饰器里的 inner
    def inner(*args,**kwargs):
        print("额外增加了一道 锅包肉")
        func(*args,**kwargs)
        print("锅包肉 38元")
    return inner

def wrapper1(func): # warpper2装饰器里的 inner
    def inner(*args,**kwargs):
        print("额外增加了一道 日魔刺生")
        func(*args,**kwargs)
        print("日魔刺生 白吃")
    return inner

def wrapper2(func):  # 被装饰的函数foo
    def inner(*args,**kwargs):
        print("额外增加了一道 麻辣三哥")
        func(*args,**kwargs)
        print("难以下嘴")
    return inner

@auth        # 1           7
@wrapper1    #   2       6
@wrapper2    #    3    5
def foo():   #      4
    print("这是一个元宝虾饭店")



# foo = wrapper2(foo) # inner = wrapper2(foo)
# foo = wrapper1(foo) # inner = wrapper1(inner)
# foo = auth(foo)     # inner = auth(inner)
# foo()               # auth里边的inner()
```

递归：不断调用自身，用函数实现

1. 死递归：不断调用自身
2. 满足递归的两个条件：
    1. 不断调用自身
    2. 有明确的终止条件
3. 最大深度：官方说明是1000层，实际测试998或者997
4. 例题：

```
def age(n):
    if n == 1:
        return 18
    else:
        return age(n-1)+2
print(age(3))        #3代表执行三次
```

拆解：

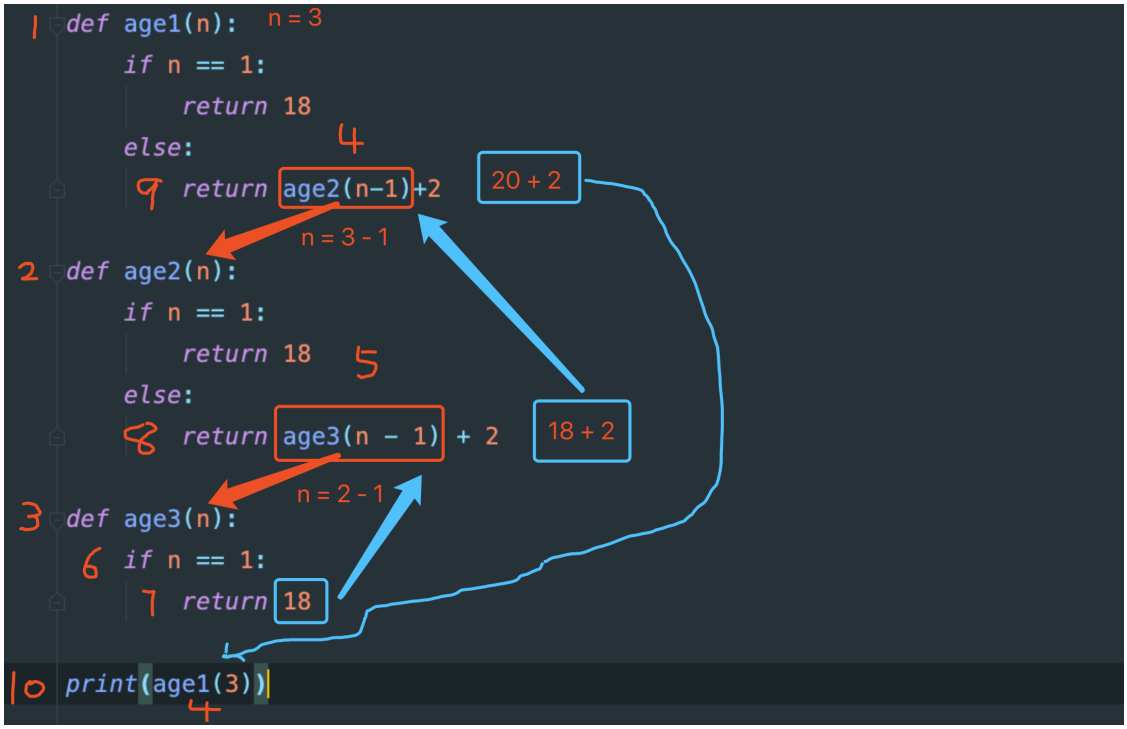
```
 if n == 1:
        return 18
    else:
        return age2(n-1)+2

def age2(n):
    if n == 1:
        return 18
    else:
        return age3(n - 1) + 2

def age3(n):
    if n == 1:
        return 18

print(age1(3))
```

流程图：
图中红色箭头是递的过程，蓝色箭头是归的过程


1 def age1(n): n = 3
if n == 1:
return 18
else:
4
9 return age2(n-1)+2
20 + 2
n = 3 - 1
2 def age2(n):
if n == 1:
return 18
5
else:
8 return age3(n - 1) + 2
18 + 2
n = 2 - 1
3 def age3(n):
6 if n == 1:
7 return 18
10 print(age1(3))
4