# python营

## python计算基础

### 几种组合

- list列表：用方括号来定义的，元素可以是相同或不同类型，列表对象是可变的。
- tuple元组：用圆括号来定义的，元素可以是相同或不同类型，tuple对象是不可变的。
- dict字典：用大括号定义的，每一项用逗号隔开，每一项都是key:value，key之间是不可以重复的。
- set集合：用大括号定义，会过滤掉重复的项，相当于dictionary的key不能重复。

```
list1=[1,'2',True]
tuple1=(1,'2',True)
dict1={'name':'Tom','age':18,'gender':'male'}
set1={1,2,3,4,5,5,5}  >>运行>>  set1={1,2,3,4,5}
```

### 备注

```
##############  range ###################
range:是一个生成函数，会生成0开始的值
range(10)  >>  range(0,10)
强制转换
list(range(10))  >>  [0,1,2,3,4,5,6,7,8,9]
list(range(3,10))  >>  [3,4,5,6,7,8,9]
list(range(3,10,2)  >>  [3,5,7,9]

##############  for   ###################
for i in xxx:
    循环内容
//in的内容不一定是数，i是代表xxx的元素
也可以是各种奇奇怪怪的东西如上面的list1，里面甚至还有True
```

---

## python基础

### 字符串进阶

#### 字符串的索引和切片

| M | O | L | L | Y |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| -5 | -4 | -3 | -2 | -1 |

- 切片的语法：[start:end:step]，三个参数都有默认值，默认截取方向从左到右的。start默认0；end默认字符串末尾元素；step默认1。如果切片的步长是负值，则截取方向是从右往左。特别注意的是start:end是一个左闭右开区间。

```
name='mollymolly'
name[-4]  >>  'o'
name[1:7:2]  >>  'olm'  #左闭右开
name[::-1]  >>  'yllomyllom'
name[8:2:-1]  >> 'llomyl'
```

```
# 如果字符串以'p'结尾，则打印
list_string = ['apple','banana_p','orange','cherry_p']
for fruit in list_string:
```

```
    if fruit[-1] == 'p':

    或者  if fruit.endswith('p'):
    这种的好处是结尾可以多几个字，如p改成pr上面那种写法就有问题。

        print(fruit)
>>>
banana_p
cherry_p
```

**字符串常用函数（计数、查找、拆分、替换）**

- len长度：返回字符串的长度。
- count计数：显示自定字符在字符串中的个数（重叠的不会计数，如abcabca里面找abca只会找到前面的1个abca）。

```
article = 'Disney and Marvel’s upcoming superhero epic should light the box office on fire when it laun
```

article.count('and') >> 3 三个 article.find('and') >> 7 返回了第一个and的首位下标 'and' in article >> True 判断是否有'and'在上文中的语句

```
+ help() ：输入函数可以返回定义。
+ find查找功能：返回从左第一个指定字符的索引，找不到返回-1。
+ index查找：返回从左第一个指定字符的索引，找不到报错。
+ startswith和endswith。
+ 更复杂的查找和拆分可以搜索“正则表达式”。
```python
my_string = 'hello_world'
my_string.find('o')  >>  4
my_string.index('o')  >>  4
my_string.find('a')  >>  -1
my_string.index('a')  >>  ValueError
my_string.startswith('hello')  >>  True 判断开头是否是它
my_string.endswith('wor_')  >>  False
```

- split：字符串的拆分，按指定内容进行分割。
- 
  - replace：字符串的替换，从左到右替换指定的元素，可以指定替换的个数，默认全部替换。replace完记得赋值回去，这个不是直接影响原字符串的。

```
>>my_string = 'hello_world'
>>my_string.split('_')    #拆分
>>>
['hello', 'world']
同理上面的article拆分单词可以用article.split(' ')来得到单词列表
>>my_string.replace('_',' ')   #替换
>>>
'hello world'
################指定替换##################
>>my_string="one one one two two three"
>>my_string.replace('one','one'.upper(),2)  #替换前两个为大写形式
>>>
"ONE ONE one two two three"
```

- 字符串标准化【 .strip() 】：默认去除两边的空格、换行符之类的，去除内容可以指定。常用于cv工程师。
- 字符串的变形：upper()、lower()、capitalize(//首字母大写)

**字符串的格式化输出**
```

| 格式 | 描述 |
| --- | --- |
| %% | 百分号标记 |
| %c | 字符及其 ASCII 码 |
| %s | 字符串 |
| %d | 有符号整数(十进制) |
| %u | 无符号整数(十进制)已过时 |
| %o | 整数(八进制) |
| %x | 整数(十六进制) |
| %X | 整数(十六进制大写字符) |
| %e | 浮点数字(科学计数法) |
| %E | 浮点数字(科学计数法，用 E 代替 e) |
| %f | 浮点数字(用小数点符号) |
| %g | 浮点数字(根据值的大小采用%e 或%f) |
| %G | 浮点数字(类似于%g) |
| %p | 指针(用十六进制打印值的内存地址) |
| %n | 存储输出字符的数量放进参数列表的下一个变量中 |

https://blog.csdn.net/weixin_44251004

```
name = 'Molly'
hight = 170.4
score_math = 95
score_english = 89
print('我叫%s，身高是%d cm， 数学成绩%.2f分,英语成绩%d分' % (name, hight, score_math, score_english))

注意%后要加括号表示这部分对应过去
```

- format

```
例子
print('我{}，身高是{:d} cm， 数学{:.2f}分,英语{}分'\
.format(name, int(hight), score_math, score_english))
```

'Hello, {0}, 成绩提升了{2:.1f}分，百分比为 {1:.1f}%'
.format('小明', 6, 17.523)

> 'Hello, 小明, 成绩提升了17.5分，百分比为 6.0%'

提高可读性
'Hello, {name:}, 成绩提升了{score:.1f}分，百分比为 {percent:.1f}%'
.format(name='小明', score=6, percent = 17.523)

```
+ **f-string: py3.6加入了该函数，很方便，可读性好**
```python
name = 'Molly'
hight = 170.4
score_math = 95
score_english = 89
print(f"大家好！我叫{name.lower()}，我的身高是{hight:.3f} cm， 数学成绩{score_math}分,英语成绩{score_english}分")

>>>
大家好！我叫molly，我的身高是170.400 cm， 数学成绩95分,英语成绩89分
```
```

## list进阶

### list常用函数

- append：在末尾添加新的元素
- insert：在指定位置前添加元素，如果指定的下标不存在，那么就是在末尾添加
- extend：将后者中的元素复制到前者的末尾

```
list1 = ['a','b','c','d','e','f']
list1.append('g')
>>>
['a','b','c','d','e','f','g']
```

print(list1) list1.insert(2, 'ooo')
print(list1)

> ['a','b'.'ooo','c','d','e','f','g'] list1.insert(-2, 'ooo')
>
> ['a', 'b', 'c', 'd', 'e', 'ooo', 'f', 'g'] 注意这些操作都是改变了list的，不只是单纯的输出而已。和list1+list2不一样。

```
```python
list1=['a','b']
list2 = ['z','y','x']
list1.extend(list2) #合并两个list   list2中仍有元素
print(list1)  >>  ['a', 'b', 'z', 'y', 'x']
print(list2)  >>  ['z','y','x']
list1=['a','b']
list1.append(list2)
print(list1)  >>  ['a','b',['z','y','x']]
```
```

### count计数和index查找

```
list1 = ['a','b','a','d','a','f']
print(list1.count('a'))   >>  3
print(list1.index('a'))  >>  0
print('a' in list1)   >>  True
```

### 删除元素

- pop()：弹出第几个元素，默认最后一个（即默认-1），弹出来的元素送返回值用。
- remove()：直接删除元素，无返回值。默认删除按顺序第一个出现的对应元素。

```
list1 = ['a','b','a','d','a','f']
x=list1.pop(3)
print(list1)  >>  ['a', 'b', 'a', 'a', 'f']
print(x)   >>  d  无引号
list1.remove('a')
print(list1)  >>  ['b', 'a', 'a', 'f']
```

### 列表生成式

- 题1：要求列表中每一项都加1

```
传统方法
list_1 = [1,2,3,4,5]
for i in range(len(list_1)):
  list_1[i] += 1
```

列表生成式pythonic方法，简洁 [n+1 for n in list_1]

```
+ 例题
```py
1. 求1~10之间所有数的平方
[(n+1)**2 for n in range(10)]

2. 求1~10之间所有数的平方，并构成字符串列表
[str((n+1)**2) for n in range(10)]

3. 在list1中所有元素前加上'app_'
list1=['a','b','c']
['app_%s'%n for n in list1]

4. 3的f-string解法
[f'app_{n}' for n in list1]
```

- 题2：获得列表中的每一个是偶数的项

```
传统方法
list_1 = [1,0,2,3,4,5]
list_2 = []
for i in range(len(list_1)):
  if list_1[i] % 2 ==0:
      list_2.append(list_1[i])
>>>
[0,2,4]
```

列表生成式方法 list_1 = [1,2,3,4,5] [n for n in list_1 if n%2==0] >> [2,4] //最前面的n是操作的意思 [n+1 for n in list_1 if n%2==0] >> [3,5]

```
+ 例题
```py
找出字符串宏所有以sv结尾的元素
list_2 = ['a','b','c_sv','d','e_sv']
[s for s in list_2 if s.endswith('sv')]

取两个list的交集
list_a=[1,3,6,7,32,65,12]
list_b=[2,6,3,5,12]
[i for i in list_a if i in list_b]

找出在list1但不在list2中的元素
[i for i in list_a if i not in list_b]
再要求要奇数
[i for i in list_a if (i not in list_b) and (i%2==1)]

列表生成式中的循环嵌套（两层其实就很少用到了）
>> [m+n for m in 'ABC' for n in 'XYZ']
>>>
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

## 生成器

- 引入：通过列表生成式，我们可以直接创建一个列表，但是受到内存限制，列表容量肯定是有限的。而且创建一个包含100万元素的列表不仅占用很大的储存空间，如果我们仅仅需要访问前面几个元素，那后面的绝大多数元素占用的空间都白白浪费了。
- 生成器表达 ```py

1. 像列表生成式一样 g=(x*x for x in range(10))

2. 1 访问--next() next(g) >> 点一下动一下出一个值，省空间

3. 2 访问--for循环 for n in g: print(n)

4. 基于函数 def reverse_print(string): print(strng[::-1]) #反输出 return 'successful!'

> ret = reverse_print('abcd')
>
> dcba //其中ret='successful!'

```
+ yield：抛出，但是可以继续迭代，不同于return完就结束

### 异常和错误处理
+ 常见的异常是错误输入，可以通过检查用户输入来做相应处理。
```py
list1 = [1,2,3,4,'5',6,7,8]
n=1
for i in range(len(list1)):    检查是否非法，非法就跳过
    if type(list1[i]) != int:
        continue
    print(list1[i])
    list1[i]+=1
```

- try语句
- - try 的工作原理是，当开始一个 try 语句后，Python 就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try 子句先执行，接下来会发生什么依赖于执行时是否出现异常。
- - 如果当 try 后的语句执行时发生异常，Python 就跳回到 try 并执行第一个匹配该异常的 except 子句，异常处理完毕，控制流就通过整个 try 语句（除非在处理异常时又引发新的异常）。
- - 如果在 try 后的语句里发生了异常，却没有匹配的 except 子句，异常将被递交到上层的 try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。

  ```
  try:
  <语句>          #运行别的代码
  except <名字>:
  <语句>          #如果在try部份引发了'名字'异常
  except <名字>，<数据>:
  <语句>          #如果引发了'名字'异常，获得附加的数据
  else:
  <语句>          #如果没有异常发生
  finally:
  <语句>          #有没有异常都会执行
  ```

- assert 语句：用于调试时判断，输出错误
- logging 语句：调试时用

---

# python函数基础

## 函数

- 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。

**定义函数及调用**

- 嵌套的例子

```
## 函数的嵌套调用
def worker(s):
  rst = 10 / float(s)
  return rst
```

def group_leader(s): rst = worker(s) * 2 return rst

def CTO(s): return group_leader(s)

CTO('4') #开始按照上面套娃

> 5.0

```
#### 参数传递
+ **位置参数**：位置参数是最简单的一种调用方式，位置参数必须以正确的顺序传入函数、数量必须和声明时一样。
+ **缺省参数**：调用函数时，缺省参数的值如果没有传入，则被认为是默认值。但也可以给默认参数赋值
```py
def student_name_and_age(name, age='不愿透露'):
    "设置默认参数"
    print('姓名：%s 年龄 %s' %(name, age))

>> student_name_and_age('张三')
>>>
姓名：张三 年龄 不愿透露

>> student_name_and_age('张三',18)
>>>
姓名：张三 年龄 18
```

- **可变参数**：即传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。定义的方式是在参数前面加*号。

```
def all_student_names(*names):
  for name in names:
      print('姓名：', name)
```

> all_student_names('张三','李四','王五')
>
> <class 'tuple'> 说明参数变成元组 姓名： 张三 姓名： 李四 姓名： 王五

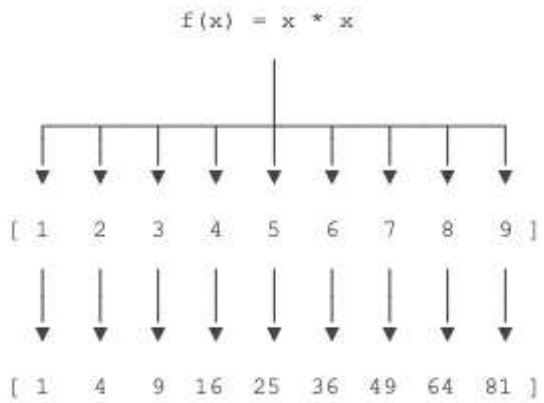
```
+ **关键字参数**：关键字参数允许传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。定义的方式是参数前加
```

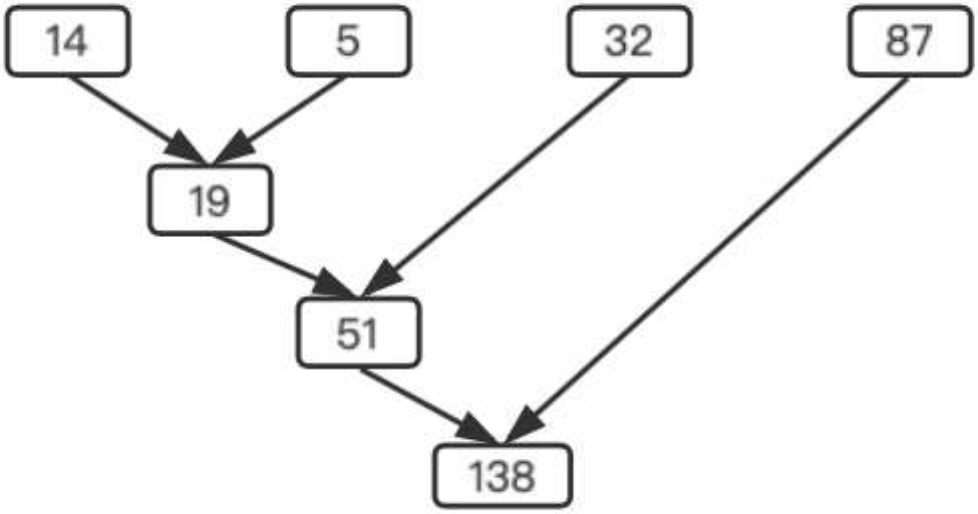
def student_info(name, age, **kw): print(type(kw)) print(f'我的名字叫：{name},年龄：{age},其它信息：{kw}')

> student_info('张三', 18, height=180)
>
> <class 'dict'> 我的名字叫：张三,年龄： 18,其它信息：{'height': 180}

```
+ +  **甲方可随时加减需求的例子**
```
```

def score_info(name, **kw): if '语文成绩' in kw: print(name, '的语文成绩', kw['语文成绩']) if '数学成绩' in kw: print(name, '的数学成绩', kw['数学成绩']) 加科目在这里加

def person_info(name, age, **kw): print('姓名：', name, '年龄',age) score_info(name, **kw)

score_cfg = {'语文成绩':65, '数学成绩':60} person_info('张三', 18, **score_cfg)

> 姓名： 张三 年龄 18 张三 的语文成绩 65 张三 的数学成绩 60

```
+ **命名关键字参数**：如果要限制关键字参数的名字，就可以用命名关键字参数。用于提醒自己这是什么参数，迭代的时候才不会忘目前版本
```

def print_person_info(name, age, *, height, weight): print('我的名字叫：', name, '年龄：', age,'身高', height, '体重', weight)

> print_person_info('张三', 18, height=180, weight=75)
>
> 我的名字叫： 张三 年龄： 18 身高 180 体重 75

```
+ 参数的组合：python中定义函数，可以是必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用，但是多
```

def student_info(name, age=18, *books, city, **kw): # 如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符了： print('我的名字叫：', name, '年龄：', age,'其它信息：',kw) print('来自：', city) for book in books: print('我有',book,'书')

> student_info('张三', 18, '语文','数学','计算机', city='北京', height=180, weight=75)
>
> 我的名字叫： 张三 年龄： 18 其它信息： {'height': 180, 'weight': 75} 来自： 北京 我有 语文 书 我有 数学 书 我有 计算机 书

```
#### 变量作用域和global变量
+ 局部变量
+ 全局变量
+ 函数优先使用局部变量，在没有局部变量的情况下使用全局变量。
+ 定义在函数内部的变量，在函数结束之后变量自动消亡。
```

def change_my_name(): global name 声明全局变量 print('我的名字曾经是', name) name = '李四' print('我的名字现在是', name)

name = '张三' change_my_name() name

> 我的名字曾经是 张三 我的名字现在是 李四 '李四'

```
#### lambda匿名函数
+ python使用lambda来创建匿名函数
+ lambda只是一个表达式，函数体比def简单很多
+ lambda的主体是一个表达式，不是代码块，仅仅能在lambda表达式中封装有限的逻辑。
+ lambda函数拥有自己的命名空间，且不能访问自有参数列表外或全局命名空间里的参数。
+ 虽然lambda函数看起来只有一行，却不等同于C或CPP的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。
```

匿名

> (lambda arg1,arg2:arg1+arg2)(1,2)
>
> > 3 常给上一个名字 add = lambda arg1,arg2:arg1+arg2 add(1,2) 3 #数字转换字符串 int2str=lambda x:str(x) int2str(5) '5'

```
#### 高阶函数
+ print本身就是一个内建函数（即不用自己建）
+ 函数名就是指向函数的变量
+ 函数名也可以作为一个变量传入其他函数
+ 一个函数可以接收另一个函数作为参数，这种函数就称为高阶函数。就是说在编写对应关系的时候不知道f是什么，就留个位置给它。
+ 例子
```py
def func_x(x, f):
    return f(x)

>> func_x(-1, abs)
>>> 1
```
```
def func_xy(x, y ,f):
    return f(x,y)
add = lambda x,y : x+y
mul = lambda x,y : x*y
SS = lambda x,y : x**2 + y**2

>> print(func_xy(1,2,add))
>> print(func_xy(1,2,mul))
>> print(func_xy(1,2,SS))

>>> 3  2  5
```

- 两个python内置的常见的高阶函数map/reduce
- 
  - **map()函数**：接收两个参数，一个是函数，一个是Iterable（可迭代的对象，list，for等），map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。




- 
- 
  - **reduce**：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，以此类推，最后得到一个结果（区别于map得到可迭代的对象）。注意：从python3开始，reduce函数纳入functools包内，使用前要import出来。
```




- 
- 
  - 例子：把列表每一项都平方

```
先设一个映射
fx=lambda x:x**2
```

ls=[1,2,3,4,5,6,7,8,9] #定义列表 ms=[] #来一个空表

方法1：使用循环来实现 for i in ls:
ms.append(fx(i))

方法2：使用map函数 rst=map(fx,ls) #对每一个ls都执行fx变换 list(rst) #直接写map(fx,ls)的话得到的是map对象，list给强制类型转换一下

```
```py
int2str=lambda x:str(x)
rst=map(int2str,ls)
list(rst)

>>> ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- 例子：累乘

```
from functools import reduce
mul_xy=lambda x,y:x*y
reduce(mul_xy,[1,3,5,7,9])
```

> 945

```
+ 高阶函数**sorted**：排序也是在程序中常用到的算法，无论是使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小，如果是数字
```py
简单排个序
sorted([36, 5, -12, 9, -21])
>>> [-21, -12, 5, 9, 36]

倒序排序(reverse：颠倒)
sorted([36, 5, -12, 9, -21], reverse=True)
>>> [36, 9, 5, -12, -21]

高阶函数：按绝对值排序key指定的函数将作用于list的每一个元素上，
```

```
并根据key函数返回的结果进行排序
sorted([36, 5, -12, 9, -21], key=abs)
>>> [5, 9, -12, -21, 36]

map函数，每一个都取绝对值然后按照绝对值去排序
list(map(abs, [36, 5, -12, 9, -21]))
>>> [36, 5, 12, 9, 21]
```

- 例子

```
将点分别按x坐标排序 y坐标排序 和0点距离排序
points = [(5,2), (7,3), (3,4),(1,1),(2,6)]
f_x = lambda x:x[0]  #x坐标
sorted(points, key=f_x)
>>> [(1, 1), (2, 6), (3, 4), (5, 2), (7, 3)]
f_y = lambda x:x[1]
sorted(points, key=f_y)
>>> [(1, 1), (5, 2), (7, 3), (3, 4), (2, 6)]
f_r = lambda x:x[0]**2+x[1]**2
sorted(points, key=f_r)
>>> [(1, 1), (3, 4), (5, 2), (2, 6), (7, 3)]
```

## 闭包和装饰器

### 闭包

- 解释：python中的闭包从表现形式上定义（解释）为：如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure).
- 注意：返回的函数并没有立刻执行，而是直到调用了f()才执行。
- 如下（几乎不会用到的反面例子）

```
def count():
  fs = []
  for i in range(1, 4):
      def f():
          # print(id(i))
          return i*i
      fs.append(f)  #这个fs没有把变化中的i传给f，因为它得到的i是闭包count()的i，是循环完的3了。
  return fs
```

f1,f2,f3=count() 这里访问的是闭包里面的i，但是闭包里面的i已经是3了 print(f1()) >> 9 print(f2()) >> 9 print(f3()) >> 9

```
```py
解决方法：return g的闭包包含到下面的f(j)
ef count():
    def f(j):
        def g():
            # print(id(j))
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs

f1, f2, f3 = count()
print(f1())
```
```

```
print(f2())
print(f3())
```

- 结论：在返回闭包时，返回函数不要引用任何循环变量或者后续会发生变化的变量。
  **装饰器**
- 解释：字面理解，就是“装饰”python的工具，使得代码更具有python简洁的风格，换句话说，它是一种函数的函数，因为装饰器传入的参数就是一个函数，然后通过实现各种功能来对这个函数的功能进行增强。
- 函数名.__name__：返回函数名
- 装饰器最简单的运用：包装函数，在已有函数的基础外添加语句包装成一个新函数，可以用于原有函数权限过高不能修改只能调用，但是需要其他语句的时候。
- 装饰器最大优势是解决重复性操作，主要应用场景有：计算函数运行时间，给函数打日志，类型检查等。
- 例子

```
# 装饰器输入一个函数，输出一个函数
def print_working(func):
  def wrapper():
      print(f'{func.__name__} is working...')
      func()
  return wrapper
```

def worker1(): print('我是一个勤劳的工作者！')

worker1 = print_working(worker1) worker1()

> worker1 is working... 我是一个勤劳的工作者！

```
+ 而为了简便，python中常用@来在定义的时候启动装饰器
```py
@print_working
def worker1():
    print('我是一个勤劳的工作者！')

>> worker1()
>>>
worker1 is working...
我是一个勤劳的工作者！
```

- 带参数的装饰器例子

```
日志
def arg_decorator(func):  #它才是最大的那个函数
  def wrapper(*args, **kw):
      print(f'{func.__name__} is working...')
      func(*args, **kw)  #注释掉的话就不会有调用下面函数的机会
  return wrapper
函数
# 带参数的装饰器
@arg_decorator
def student_info(name, age=18, *books, **kw):
  print(f'我的名字叫{name}, 今年{age}岁，我有很多本书：')
  for book in books:
      print(book)
  print(kw)
调用
>> student_info('Molly',18, '语文书','数学书',height=170)
>>>
student_info is working...
我的名字叫Molly, 今年18岁，我有很多本书：
```
```

```
语文书
数学书
{'height': 170}
```

- 实例

```
from functools import wraps
#使用了装饰器，切面的效果，横切，@一次就装饰一次
def mylog(func):
  #@wraps(func)   #有没有这个结果都一样
  def with_logging(*args):
      print(func.__name__+" was called")
      print(args)    #传进来的参数打印出来
      return func(*args)   #末尾
  return with_logging   #执行的动作
```

@mylog def add(a,b): #这装饰到def mylog(func)中的func中 print(a+b)

@mylog def mins(a,b): print(a-b)

@mylog def mul(a,b): print(a*b)

@mylog def div(a,b): print(a/b)

add(1,2) mins(1,2) mul(1,2) div(1,2)

> add was called (1, 2) 3 mins was called (1, 2) -1 mul was called (1, 2) 2 div was called (1, 2) 0.5

```
### 偏函数（也是高阶函数）
+ partial函数：通过设定参数的默认值，降低函数调用的难度。需要先from functools import partial才能用
```py
def student_info(name, age, city):
    print(f'我的名字叫{name}, 今年{age}岁，来自{city}')
student_info_beijing = partial(student_info, city='北京')
>> student_info_beijing('Molly',18)
>>>
我的名字叫Molly, 今年18岁，来自北京
>> student_info_beijing('Molly',18, city='上海')
>>>
我的名字叫Molly, 今年18岁，来自上海

 另一个例子
def add_xy(x,y):
    return x+y
add_100 = partial(add_xy, y=100)
add_100(10)
>>> 110
```

## 模块

- import：类似于#include
- 简写：如from functools import partial as pr 这样子
- 习惯上有：import numpy as np和import pandas as pd
- 安装第三方模块：pip install/uninstall
```

# python面对对象

## 预备知识

### 命名空间

- 内置命名空间（Built-in names）： 用于存放Python 的内置函数的空间，比如，print，input等不需要定义即可使用的函数就处在内置命名空间。
- 全局命名空间（Global names）：模块中定义的名称，记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。
- 局部命名空间（Local names）：函数中定义的名称，记录了函数的变量，包括函数的参数和局部定义的变量。在函数内定义的局部变量，在函数执行结束后就会失效，即无法在函数外直接调用函数内定义的变量。
- 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。
- 命名空间的生命周期： 命名空间的生命周期取决于对象的作用域，如果对象执行完成，则该命名空间的生命周期就结束。

```
在全局命名空间中给变量a赋值，令a=1。
a = 1
```

在定义函数“function1”的同时就为其创建了一个独立的局部命名空间。 def function1(): 变量b在局部命名空间中被赋值“a+1”。 b = a + 1 此处所使用的print函数就是源自Python的内置命名空间。 print(b)

执行function1时，计算机会优先找到function1局部命名空间中的“b=a+1”， 然后再找到全局命名空间中的“a=1”， 最后找到内置命名空间中的print将b的值打印出来。

function1()

```
#### 作用域
+ 作用域就是一个Python程序可以直接访问命名空间的正文区域。
+ 4种类型：
+ + Local：在程序的最内层，包含局部变量，比如，一个函数的内部。

+ + Enclosing：包含了非局部(non-local)也非全局(non-global)的变量。比如，两个嵌套函数，函数（或类）A里面又包含了函数B，那
+ + Global：当前脚本的最外层，比如，当前模块的全局变量。
+ + Built-in:  包含了内建的变量/关键字等，比如，int。
#### 排序
+ sorted()：可对所有可迭代对象进行排序
+ sort()：是应用在list上的方法
+ list的sort方法返回的是对已经存在的列表进行操作，而内建函数sorted方法返回的是一个新的list，而不是在原来的基础上进行的操作。
+ 语法：sorted(iterable, key=None, reverse=False)，其中 iterable：可迭代对象。key：主要是用来进行比较的元素，只有一个
#### id函数
+ id()函数返回对象的唯一标识符，标识符是一个整数。Cpython中的id()可用于获取对象的内存地址
+ 语法：id(object)，其中object是需要获取标识符的对象
#### dir函数
+ dir()函数不带参数时，可返回当前范围内的变量、方法和定义的类型列表；带参数时，可返回参数的属性、方法列表。如果参数包含方法dir
+ 语法：dir(object)
#### split函数
+ split()函数可通过指定分隔符对字符串进行分割，如果参数num有指定值，则分隔num+1个子字符串。
+ 语法：str.split("sign", num)，其中：sign：分隔符，用于指定对字符串进行分割的单位，常见的分隔符有空格、逗号等。num：分割
```

定义一个中间有两个空格的字符串。 str = "abc def ghi" 查看并对比使用split()函数之后，str的变化。

> print(str, str.split(" "), str.split(" ", 1))
>
> abc def ghi ['abc', 'def', 'ghi'] ['abc', 'def ghi']

```
#### threading模块
```

```
+ threading，基于线程的并行，这个模块在较低级的模块 _thread (底层多线程 API)基础上建立较高级的线程接口。
+ 该模块定义了以下函数：

+ + threading.active_count()：返回当前存活的线程类 Thread 对象。返回的计数等于 enumerate()返回的列表长度。
+ + threading.current_thread()：返回当前对应调用者的控制线程的 Thread 对象。如果调用者的控制线程不是利用 threading 创建
+ + threading.get_ident()：返回当前线程的 “线程标识符”。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，
#### zipfile模块
+ ZIP 文件格式是一个常用的归档与压缩标准。 这个模块提供了创建、读取、写入、添加及列出 ZIP 文件的工具。 任何对此模块的进阶使用
***
### 面对对象（上）
#### 例子引入
+ 4个运动员的数据文件，分数（时间）的格式有-和：和 . 三种形式
+ 如：james.txt 2-34,3:21,2,34,2.45,3.01,2:01,2:01,3:10,2-22
+ 先划分文件内容（用逗号），编写了get_coach_data函数，其中 ： ① filename为文件路径 ② f表示文件对象 ③  f.realine()表示
```py
# 读取文件内容，按逗号进行切分
def get_coach_data(filename):
    with open(filename) as f:
        line = f.readline()
    return line.strip().split(',')

# 输出读取文件的结果
times = get_coach_data('mywork/james.txt')
print('读取文件james.txt后的结果：\n')
print(times)
```

- 再编写函数将时间的格式处理一下，sanitize函数：① time_string为时间数组 ② splitter是根据原数据的格式来确实分钟和秒的分隔符 ③ (mins,secs) = time_string.split(splitter)以splitter分隔符切分每个时间数据到两个变量中mins,secs ④ return (mins+'.'+secs)将分钟和秒以字符串点进行连接

```
# 数据格式标准化
def sanitize(time_string):
  if '-' in time_string:
      splitter = '-'
  elif ':' in time_string:
      splitter = ':'
  else:
      return (time_string)
  (mins,secs) = time_string.split(splitter)
  return (mins+'.'+secs)
```

- 一句话搞定数据标准化，排序和去重

```
print(sorted(set([sanitize(t) for t in james_times]))[0:3])
```

**类的使用**

- 为了便于维护，引入类

```
class Athlete: #可以理解成在计算机中开辟athlete的大块内存
  def __init__(self,a_name,a_dob=None,a_times=[]): #先执行，把变量确定
      self.name = a_name  #局部变量赋值给类的属性：点name，
      self.dob = a_dob
      self.times = a_times

  def top3(self):
      return sorted(set([sanitize(t) for t in self.times]))[0:3]

  def sanitize(self,time_string):  #函数在类里面叫做类的方法
      if '-' in time_string:
          splitter = '-'
```
```

```
        elif ':' in time_string:
            splitter = ':'
        else:
            return (time_string)
        (mins,secs) = time_string.split(splitter)
        return (mins+'.'+secs)
# 创建Athlete类的对象James，类的对象无非就是多些空间，类是创建，用是用对象，对象.属性（变量）的方法来调用
james = Athlete(james_name,james_dob,james_times) #等号右边叫做类的实例化，等号左边叫做类的实例化的对象，也叫实例
print('姓名：%s,生日：%s,最快的3次成绩：%s' %(james.name,james.dob,james.top3()))
```

- 类的好处：1. 降低复杂性->更少的bug->提高可维护行 2. 类可以将数据与函数绑定在一起，使代码模块化 3. 调用数据和函数，使用对象名.的方式，使代码更加优雅
- 代码通常称为类的方法，数据通常称为类的属性，实例化对象称实例。
- **定义类的方法**:
- 
  - 第一部分：class定义类的关键字，Athlete符合python标识符命名规则，：表示类内容的开始
    def **init**(self,a_name,a_dob=None,a_times=[]):
- 
  - 第二部分：def定义函数的关键字，**init** 方法是一个特殊方法会在实例化对象时自动调用，我们会在这个方法中对数据进行赋值。self作为类中函数的第一个参数，方便该方法调用该类的其他属性和方法。
- 
  - 第三部分：自定义的属性和方法
- 使用类：
- 
  - 创建对象
    对象名 = 类名(参数)
- 
  - 使用.调用类的方法和属性
    对象.属性名
    对象.方法名() ```py 类属性

1.定义：在 **init__之上，或者说在类的范围内与方法同等级别，书写变量名=值 2.调用：类名.类属性 class Athlete: #运动员集训了，要买东西的同学要把地址改一下 address = '中国足球协会训练基地xx街xx号' #地址一样，所有变量（人）共享这个数据 def __init**(self,a_name,a_dob=None,a_times=[]):

类方法：所有对象共享的方法 1.定义：类方法在定义时，使用@classmethod标记 2.调用：类名.类方法 或 对象.类方法 末尾： #类方法，所有的对象共用的方法 @classmethod #类方法的方法 def changeAddress(self): self.address = '中国田径训练基地xx街xx号' Athlete.changeAddress() #集训队换地方了 print(julie.address) >>> 中国田径训练基地xx街xx号

```
+ 类中的函数私有化：在属性和方法名前加"__"两个下划线，若需要调用要在类中去叠加函数调用。只能通过类中的方法来调用私有的属性和方法
+ 例子
```py
#实例：狗
class Dog:
    def __init__(self):
        self.color='white'
        self.ty='哈士奇'
        self.sex='wm'
        self.age=4
    def bark(self):
        print('狗叫')
    def run(self):
        print('runrunrun')
    def sleep(self):
        print('huhuhu')
dog=Dog()
print(dog.color)
print(dog.age)
dog.bark()
dog.run()
>>>
```
```

```
>>>
white
4
狗叫
runrunrun

#严谨一点，不是所有的狗都这样子
class Dog:
    def __init__(self,color,ty,sex,age):
        self.color=color
        self.ty=ty
        self.sex=sex
        self.age=age
    def bark(self):
        print('狗叫')
    def run(self):
        print('runrunrun')
    def sleep(self):
        print('huhuhu')
dog=Dog('white','哈士奇','m',5)
print('颜色:%s 种类: %s 性别: %s 年龄: %d'%(dog.color,dog.ty,dog.sex,dog.age))
print(dog.age)
dog.bark()
dog.run()

>>>
颜色:white 种类: 哈士奇 性别: m 年龄: 5
5
狗叫
runrunrun
```

## 面对对象（下）

- **面向对象的三个特性：封装（函数、类）、继承、多态**
  **类的继承**
- 定义：class 子类名(父类名):
- 
  - 情况1，如果子类有新增的属性，那么需要在子类**init方法中，调用父类的__init**
- 
  - 情况2，如果子类没有新增的属性,子类不需要写__init__方法
- 使用：对象名 = 子类名(参数)
- 继承的好处：代码重用，升级功能（重写），新增功能（新的方法）

```
#定义橄榄球运送员类
class Rugby(Athlete):    #rugby继承athlete，代码重用
  def __init__(self,a_name,a_bod,a_squat,a_times):    #新局部变量squat
      #调用父类__init__
      Athlete.__init__(self,a_name,a_bod,a_times)   #父类里面有这三个变量直接用
      #新来的深蹲次数
      self.squat = a_squat
  # 继承后下面两个函数就在Rugby类中，只是看不到而已
  # def top3(self):
  #     return sorted(set([self.sanitize(t) for t in self.times]))[0:3]
  # def sanitize(self,time_string):
  #     if '-' in time_string:
  #         splitter = '-'
  #     elif ':' in time_string:
  #         splitter = ':'
  #     else:
  #         return (time_string)
  #     (mins,secs) = time_string.split(splitter)
  #     return (mins+'.'+secs)
```

### 类的方法重写

- 子类方法与父类方法完全相同，子类若重写了父类的方法，则子类对象调用方法时就是调用的自己类中重写的方法。重写和覆盖是一个意思，overlight。

```
class OtherAthlete(Athlete):
  def __init__(self,a_name,a_bod,a_squat,a_times):

      Athlete.__init__(self,a_name,a_bod,a_times)

      self.squat = a_squat
  def top3(self):  #重写的函数
      return sorted([self.sanitize(t) for t in self.times])[0:3]  #少了set可以重复
```

### 多态

- 多态性：一个事物多种形态
- 多态的好处是：减少重复代码，分离经常改变的代码与不经常改变的代码，使得代码可维护性提高。

```
def print_rugby(athlete):

  print(athlete.name)
  print(athlete.dob)
  print(athlete.squat)
  print(athlete.top3())
```

### 多继承

```
class Father():
    def __init__(self):
        self.color = 'black'
    def talk(self):
        print("---爸爸的表达能力---")

class Mother():
    def __init__(self):
        self.height = 170
    def smart(self):
        print("---妈妈聪明的头脑---")

class Child(Father,Mother):
    def __init__(self):
        Father.__init__(self)
        Mather.__init__(self)

child1 = Child()
child1.talk()
child1.smart()
print(child1.color)
print(child1.height)
>>>
---爸爸的表达能力---
---妈妈聪明的头脑---
black
170
```

## 模块关系

- import sys ： 导入sys模块
- sys.path.append('work')：将模块athlete.py添加到模块搜索路径

- from athlete import * ：导入athlete模块，使用athlete模块下的所有代码
  **例子**
- 问题：如果有一个模块mymodule.py中也包含get_coach_data函数，该怎么区分这两个函数呢

```
import sys
sys.path.append('mywork') #首先把work加入到搜索路径
```

from p1.mymodule import * from p2.mymodule import * #import p1 有了上面那句这句可以不用 #import p2 p1.mymodule.demo() #p1包（文件夹）下的mymodule下的demo p2.mymodule.demo() #这样子才能从包里取出同个路径下相同函数名的函数

> p1 -- mymodule p1 -- mymodule

```
+ 例子：创建交通工具类
```py
#vehicle父类,包括speed，bike，car
class Vehicle:
    def __init__(self,speed):
        self.speed=speed
    def drive(self,distance):
        print('time:%s' %(distance/self.speed))
#测试一下
#veh=Vehicle(10)
#veh.drive(120)

class Bike(Vehicle):
    def drive(self,distance):
        print('strength:%s' %(distance/self.speed))
class Car(Vehicle):
    def __init__(self,speed,fuel):
        Vehicle.__init__(self,speed)
        self.fuel=fuel

    def drive(self,distance):
        print('fuel: %s' %(distance*self.fuel))

#bk=Bike(2)
#bk.drive(120)

#car=Car(20,0.5)
#car.drive(120)

def demo_drive(veh): #多态
    veh.drive(120)

demo_drive(Bike(2))
demo_drive(Car(20,0.5))
>>>
strength:60.0
fuel: 60.0
```
```

# 文件处理

## 简单文件操作

- open() 为bif（build in function内置函数），参数有多个，必须的是文件路径。 返回的一个文件对象。
- file.readline(),读取文件中的一行。
- 对付异常数据的处理方法：①使用异常跳过有问题的数据（pass）② 增加代码判断

- open和close的clean的写法，三行变一行，以后都这么写就行，不用open、close（这里没有错误跳过的功能）：

```
with open('路径') as f:
  for xxxx in f:
      xxx
  这个缩进下都是打开后的操作，离开这个缩进就不属于打开的范围
```

- file对象的函数列表：

```
with open('work/train_data_cor.txt') as f:
  data = f.read()
  print('整个文件\n'+data)
  f.seek(0) #指针移动到0的位置上去
  data = f.read(10)  #然后读10个
  print('读取指定大小的文件内容\n'+data)
  print(f.tell()) #返回指针当前的位置
```

- 写入文件内容：

```
f = open('work/data.txt','w') #写文件，没有的话就会创建
f.write('this is file content')
f.close()
```

open('work/data.txt','w')第一个参数文件路径，第二个参数打开文件的模式
f.write('this is file content')参数为写入的内容 f.close()关闭文件

```
### json
#### 对象转json
+ key:value
+ json就是一个中间载体
```py
import json #导入
class Athlete(json.JSONEncoder): #继承

    def __init__(self,a_name,a_dob=None,a_times=[]):
        self.name = a_name
        self.dob = a_dob
        self.times = a_times
    def top3(self):
        return sorted(set([self.sanitize(t) for t in self.times]))[0:3]
    def sanitize(self,time_string):
        if '-' in time_string:
            splitter = '-'
        elif ':' in time_string:
            splitter = ':'
        else:
            return (time_string)
        (mins,secs) = time_string.split(splitter)
        return (mins+'.'+secs)


with open('work/train_data_cor.txt') as f:
    data = f.readline().strip().split(',')
    ath = Athlete(data.pop(0),data.pop(0),data)  #实例化对象，pop出名字、生日、时间
    print(ath) #打印对象

ath_json = json.dumps(ath.__dict__) #使用到内置属性（转化为字典形式），dumps是把对象序列化成一个json字符串对象
#dumps把类变成变量
print(ath_json)
>>>
```
```

```
<__main__.Athlete object at 0x7fed2ca01f10>
{"name": "james", "dob": "2004-5-21", "times": ["2.34", "3:21", "2.34", "2.45", "3.01", "2:01", "2:01", "3:
```

### 内存中的json形式的变量保存到文件

```
with open('work/json.txt','w') as f:
    json.dump(ath_json,f) #将json对象存到文件里
```

### 读取json文件内容

```
with open('work/json.txt') as f:
    ath = json.load(f)  #加载，转化为变量
    print(ath)   #然后输出出来
>>>
{"name": "james", "dob": "2004-5-21", "times": ["2.34", "3:21", "2.34", "2.45", "3.01", "2:01", "2:01", "3:
```

## 目录访问

```
#当前路径
import os
#返回当前工作目录
current_path = os.getcwd()
print('当前路径: '+current_path)

#改变当前工作目录
os.chdir('/home/aistudio/work')
#运行mkdir命令
os.system('mkdir today') #执行命令行，创建文件夹

#检查路径
from pathlib import Path  #path模块
abs_path = os.path.abspath('')  #返回当前绝对路径
print('abs_path: '+abs_path)
print(os.getcwd())  #当前路径
Path(abs_path).exists()  #检查路径是否存在

listdir = os.listdir()  #几个小函数的运用
#返回当前路径下文件和文件夹名
print(listdir)

os.path.isdir('/home/aistudio/work/today')  #判断是否为文件夹
```

- 实例：显示work路径下的所有类型为txt的文件

  ```
  import pdb #python debug  然后在后文需要加断点的地方打pdb.set_trace()就行
  import os
  listdir = os.listdir('/home/aistudio/work')
  ```

target = [] for name in listdir: #防止文件名与文件夹名一样的情况 # print(os.path.isfile(name))

```
temp = name.split('.')
(filename,filetype) = (temp.pop(0),temp.pop(0))
if filetype == 'txt':
    target.append(name)

# print('name:%s,type:%s' %(filename,filetype))
```

print(target)

```
+ 检索路径存在需要检索的文件夹：循环介入
### pdb：单步调试
+ 略

### 进程与线程
+ 单进程
```py
#只使用进程的方式
import zipfile

print('压缩作业开始了，请您耐心等待...')

infile = 'work/loren.txt'
outfile = 'work/myarchive.zip'
f = zipfile.ZipFile(outfile, 'w', zipfile.ZIP_DEFLATED)
f.write(infile)
f.close()


print('压缩作业结束了，请问还需要帮您做什么呢？')
```

- 进程+线程

```
import threading, zipfile  #线程的包
```

class AsyncZip(threading.Thread): #自定义个父类 def **init**(self, infile, outfile): threading.Thread.**init**(self) self.infile = infile self.outfile = outfile

```
def run(self):  #以线程的方式运行
    f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
    f.write(self.infile)
    f.close()
    print('压缩完成，您要的文件在:', self.outfile)
```

background = AsyncZip('work/loren.txt', 'work/myarchive.zip')

print('压缩作业开始了，请您耐心等待...') background.start() print('我正在为您压缩，请问还需要帮您做什么呢？') background.join()
```