Python 语言

一、Python基础

1、Python 语法基础

默认情况下，Python 源码文件以 UTF-8 编码，Python 编码指定：

-*- coding: utf-8 -*-

Python 保留字即关键字，不能作为任何标识符名称。Python 标准库 keyword 模块可以输出当前版本的所有关键字。

Python 中使用单引号和双引号创建字符串，使用三引号('''或""")可以创建一个多行字符串。

反斜杠可以用来转义，使用 r 可以让反斜杠不发生转义，如 r"this is a line with \n"则\n 会显示，并不换行。

字符串可以用+运算符连接在一起，用*运算符重复。

Python 中的字符串有两种索引方式，从左往右以 0 开始，从右往左以-1 开始。

2、Python 运算符

（1）算术运算符

加（+）：两对象相加

减（-）：两对象相减

乘（*）：两个数相乘或是返回一个被重复若干次的字符串

除（/）：x 除以 y

取模（%）：返回除法的余数

幂（**）：返回 x 的 y 次幂

取整除（//）：向下取接近除数的整数

（2）比较运算符

等于（==）：比较对象是否相等

不等于（!=）：比较对象是否不相等

大于（>）：x 是否大于 y

小于（<）：是否 x 小于 y

大于等于（>=）：x 是否大于等于 y

小于等于（<=）：x 是否小于等于 y

（3）赋值运算符

赋值（=）：简单赋值运算

加法赋值（+=）：c += a 等效于 c = c + a

减法赋值（-=）：c -= a 等效于 c = c - a

乘法赋值（*=）：c *= a 等效于 c = c * a

除法赋值（/=）：c /= a 等效于 c = c / a

取模赋值（%=）：c %= a 等效于 c = c % a

幂赋值（**=）：c **= a 等效于 c = c ** a

取整除赋值（//=）：c //= a 等效于 c = c // a

（4）逻辑运算符

逻辑与（and）：x and y，如果 x 为 False，返回 x，否则返回 y 的值。

逻辑或（or）：x or y，如果 x 是 True，返回 x 的值，否则返回 y 的值。

逻辑非（not）：not x，如果 x 为 True，返回 False 。如果 x 为 False，返回 True。

（5）位运算

按位与运算符（&）：参与运算的两个值，如果两个相应位都为 1，则该位的结果为 1，否则为 0。

按位或运算符（|）：只要对应的二个二进位有一个为 1 时，结果位就为 1。

按位异或运算符（^）：当两对应的二进位相异时，结果为 1。

按位取反（~）：对数据的每个二进制位取反，即把 1 变为 0，把 0 变为 1。~x 等价于-(x+1)。

左移运算符（<<）：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补 0。

右移运算符（>>）：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。

（6）成员运算符

in：如果在指定的序列中找到值返回 True，否则返回 False。

not in：如果指定序列中没有找到值返回 True，否则返回 False。

（7）身份运算符

is：x is y，用于判断两个标识符是不是引用自一个对象，如果引用的是同一个对象则返回 True，否则返回 False。

is not：x is not y，用于判断两个标识符是不是引用自不同对象，如果引用的不是同一个对象则返回结果 True，否则返回 False。

（8）运算符优先级

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
and or not	逻辑运算符

3、Python 流程控制

（1）条件控制

Python 中 if 语句形式如下：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

（2）循环控制

Python 支持 for 和 while 循环语句，但没有 do..while 循环语句。

在 while … else 在条件语句为 false 时执行 else 的语句块。

```
count = 0
while count < 5:
   print("Hello Python3")
   count += 1
else:
   print("Hello Go")
```

如果 while 循环体中只有一条语句，可以将与 while 写在同一行中。

for 循环的一般格式如下：

```
for <variable> in <sequence>:
   <statements>
else:
   <statements>
```

当 for 循环语句代码块正常运行完时，才会运行 else 语句；如果 for 循环中使用 break 语句用于跳出当前循环体，不会正常结束 for 循环，即不执行 else 分支。

range()内置函数可以用于生成数列。

通过 seq[0:len(seq):step]可以从 seq 每隔 step 步长取一个元素构成新的序列。

Python 中 pass 空语句用于保持程序结构的完整性，pass 不做任何事情，一般用做占位语句。

（3）switch 实现

Python 不支持 switch，通过字典可以实现 switch，实现方案如下：

A、利用字典取值 get 方法的容错性，处理 switch 语句中的 default 情况。

B、设置字典的 value 为对应方法名，来代替 switch 语句中代码块。

C、为不同 key 设置相同的 value，模拟 switch 中穿透。

二、Python标准数据类型

Python3 六种标准数据类型中，Number（数字）、String（字符串）、Tuple（元组）是不可变的，List（列表）、Dictionary（字典）、Set（集合）是可变的。

1、数字

Python 数字数据类型用于存储数值，数字数据类型变量不允许改变值，如果改变数字数据类型的值，将重新分配内存空间。

Python 中 Number 有四种类型：bool、int、float、complex。bool 类型表示真假，值为 True、False；int 类型表示长整数；float 表示浮点数；complex 表示复数。

如果需要对数据内置的类型进行转换，数据类型的转换，只需要将数据类型作为函数名即可。

choice(seq)：从序列的元素中随机挑选一个元素，比如 random.choice(range(10))，从 0 到 9 中随机挑选一个整数。

randrange([start,] stop [,step])：从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为 1。

random()：随机生成下一个实数，在[0,1)范围内。

shuffle(seq)：将序列的所有元素随机排序

uniform(x,y)：随机生成下一个实数，在[x,y]范围内。

2、字符串

字符串是 Python 最常用数据类型，可以使用引号(' ，"或""")来创建字符串，其中"""用于创建多行字符串。

Python 不支持单字符类型，单字符在 Python 中作为字符串使用。

Python 访问子字符串，可以使用方括号来截取字符串。

使用索引访问字符串中字符时，如果使用负数索引，表示从字符串右侧开始进行索引，-1 表示字符串右侧开始的第一个字符。

str.count(sub，start= 0,end=len(string))：统计字符串里某个字符出现的次数。可选参数为在字符串搜索的开始与结束位置。

str.encode(encoding='UTF-8',errors='strict')：以指定的编码格式编码字符串，errors 参数可以指定不同的错误处理方案。

bytes.decode(encoding="utf-8"，errors="strict")：指定的编码格式解码 bytes 对象。默认编码为 'utf-8'。

str.endswith(suffix[，start[，end]])：用于判断字符串是否以指定后缀结尾，如果以指定后缀结尾返回 True，否则返回 False。可选参数"start"与"end"为检索字符串的开始与结束位置。

find(str，beg=0，end=len(string))：检测字符串中指定索引范围内是否包含子字符串 str。如果包含，返回匹配起始位置；如果不包含，返回-1。

index(str，beg=0，end=len(string))：如果包含子字符串返回开

始的索引值，否则抛出异常。

join(sequence)：用于将序列中的元素以指定的字符连接生成一个新的字符串。返回通过指定字符连接序列中元素后生成的新字符串。

len(str)：返回对象（字符、列表、元组等）长度或项目个数。

lstrip([chars])：用于截掉字符串左边的空格或指定字符。返回截掉字符串左边的空格或指定字符后生成的新字符串。

replace(old，new[，max])：把字符串中的 old 替换成，如果指定第三个参数 max，则替换不超过 max 次。

rfind(str，beg=0 end=len(string))：返回字符串最后一次出现的位置，如果没有匹配项则返回-1。

rindex(str，beg=0 end=len(string))：返回子字符串 str 在字符串中最后出现的位置，如果没有匹配的字符串会报异常，可以指定可选参数[beg:end]设置查找的区间。

rstrip([chars])：删除字符串末尾的指定字符（默认为空格）

split(str=""，num=string.count(str))：指定分隔符对字符串进行切片，如果参数 num 有指定值，则仅分隔 num+1 个子字符串。返回分割后的字符串列表。

startswith(substr，beg=0,end=len(string))：用于检查字符串是否是以指定子字符串开头，如果是则返回 True，否则返回 False。

strip([chars])：用于移除字符串头尾指定的字符（默认为空格）或字符序列，返回移除字符串头尾指定的字符序列生成的新字符串。

ord(c)：获取字符 c 的 ASCII 码编码值

3、列表

列表是 Python 常用数据类型，列表的数据项不需要具有相同类型。

列表元素的访问可以使用索引进行，可以指定开始索引和结束索引进行切片操作。

对列表进行解包时，接收对象个数必须与列表的元素个数相匹配。

```
a, b, c = [1, "Hello Python", True] # a, b = [1,"Hello Python", True]错误
```

del 语句可以用于删除列表的的元素。

+：组合，将两个列表进行组合后生成新的列表

*：重复，将列表的元素重复多次，生成新的列表

x in list：如果 x 在列表中返回 True，否则返回 False。

for x in list: print(x, end=" ")：对列表进行迭代

len(list) 列表元素个数

max(list) 返回列表元素最大值

min(list) 返回列表元素最小值

list(seq) 将序列转换为列表

list.append(obj) 在列表末尾添加新的对象

list.count(obj) 统计某个元素在列表内出现的次数

list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

list.index(obj) 从列表中找出某个值第一个匹配项的索引位置

list.insert(index,obj) 在列表的 index 位置插入对象 obj

list.pop(index=-1) 移除列表中位置为 index（默认为最后一个）

的元素，并返回元素的值

list.remove(obj) 移除列表中某个值的第一个匹配项

list.reverse() 反向列表中的元素

list.sort(key=None,reverse=False) 对列表进行排序。key 用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序；reverse 指定排序规则，reverse = True 降序，reverse = False 升序(默认)。

list.clear() 清空列表

list.copy() 复制列表

4、元组

元组使用小括号创建，只需要在括号中添加元素，并使用逗号隔开，元素可以是不同类型。元组可以使用下标索引来访问元组中的值。

对元组进行解包时，接收对象个数必须与元组的元素个数相匹配。

元组中的元素值是不允许修改的，但可以对元组进行连接组合。

tuple 元素不可变有一种特殊情况，当元素是可变对象时，对象内部属性是可以修改的。tuple 的不可变限制只是在一个纬度上：元素的类型。tuple 元素所保存的内容(数值或内存地址)是不允许修改的，但地址映射的对象自身是可以修改的。

```
tup1 = ("Go", "Python", [1,2,True])
tup1[2][2] = False
```

元组中的元素值不允许删除，但可以使用 del 语句来删除整个元组。

+：连接两个元组，生成新的元组。

*：重复多次元组

x in tuple：如果 x 在元组中返回 True，否则返回 False。

for x in tuple：迭代元组

len(tuple) 计算元组元素的个数

max(tuple) 返回元组中元素的最大值

min(tuple) 返回元组中元素的最小值

tuple(seq) 将序列转换为元组

5、字典

字典是一种无序的数据结构，可存储任意类型对象。键必须是唯一的，但值则不必唯一。字典中键必须是不可变的，如字符串，数字或元组；值可以取任何数据类型，可以是用户定义类型。

字典中不允许同一个键出现两次，创建时如果同一个键被赋值两次，后一个值会被记住。键必须不可变，可以用数字、字符串或元组作为键，列表不可以。

len(dict) 计算字典元素个数，即键的总数。

str(dict) 输出字典

dict.clear() 删除字典内所有元素

dict.copy() 返回一个字典的浅复制

dict.fromkeys(seq[,value]) 以序列 seq 中元素做字典的键，value 为字典所有键对应的初始值创建一个新字典。

dict.get(key[,default=None]) 返回字典中指定键的值，如果值不

存在，返回 default 值。

key in dict：如果 key 存在字典中，返回 True，否则返回 False。

dict.items() 以列表返回可遍历的(键，值) 元组数组

dict.keys() 返回一个迭代器，可以使用 list 转换为列表

dict.update(dict2) 将字典 dict2 的键值对更新到 dict 字典里

dict.values() 返回字典中所有值的迭代器，可使用 list 转换为列表

dict.pop(key[,default]) 删除字典给定键 key 所对应的值，返回值为被删除的值。key 值必须给出。 否则，返回 default 值。

6、集合

集合是一个无序的无重复元素序列，可以使用 set() 函数创建集合。

创建一个空集合必须用 set() 而不是{}，{}用来创建一个空字典。

集合不能使用索引对集合元素进行访问，运算包括差、并、交。

add(x)：将元素 x 添加到集合，如果元素存在，则不进行任何操作。

update(x)：将 x 添加到集合中，x 参数可以是列表、元组、字典等。

remove(x)：将 x 从集合中移除，如果元素不存在，则会发生错误。

len(s)：计算集合的元素个数。

clear()：清空集合。

x in s：如果 x 在集合 s 中，返回 True，否则返回 False。

copy()：返回一个集合的拷贝

set.difference(set)：返回集合的差集。

set.difference_update(set)：在原始集合上移除指定集合中存在

的元素，没有返回值。

set.intersection(set1，set2 … etc)：返回两个或更多集合中都包含的元素，即交集。

set.intersection_update(set1，set2 … etc)：在原始的集合上移除不重叠的元素，没有返回值。

set.issubset(set)：判断集合是否是指定集合的子集，如果是则返回 True，否则返回 False。

set.union(set1，set2...)：返回两个集合的并集，即包含了所有集合的元素，重复的元素只会出现一次。

7、推导式

推导式是 Python 的一种独有特性，可以从一个数据序列构建另一个新的数据序列。

（1）列表推导式

变量名 = [表达式 for 变量 in 列表 for 变量 in xxx]

```
print([i*i for i in range(1, 11)])
```

遍历出列表中内容给变量，表达式根据变量值进行逻辑运算。

变量名 = [表达式 for 变量 in 列表 if 条件]

```
print([i*2 + 1 for i in range(1, 11) if i % 2 == 0])
```

遍历列表中内容给变量，然后进行判断，符合条件的值再给表达式。

（2）字典推导式

字典推导式与列表推导式语法类似，[]替换为{}即可。

```
data = {"a": 10, "b": 20}
print({v**2: k for k, v in data.items() if v > 0})
```

（3）集合推导式

集合推导式与列表推导式语法类似，[]替换为{}即可。

```
print({x**3 for x in range(10) if x % 2 == 0})
```

三、Python包管理

1、Package

（1）Package

Python 使用 Package 管理模块，Package 通常对应一个目录，必须在目录下创建一个__init__.py，Python 才会将目录解析为包。__init__.py 本身是一个模块，模块名称与 Package 名称相同，用于标识当前目录是一个包。

（2）__init__.py 模块

__init__.py 文件可以为空，模块名称即为 Package 名称，__init__.py 文件内可以定义初始化 Package 内容，导入 Package 的内容，限制 package 内模块的导出。导入 Package 或 package 的模块、变量、函数时，__init __.py 文件会被自动执行。因此，__init__.py 文件可以实现如下应用：

A、限制本 Package 的模块导入

在__init__.py 写入内置函数__all__，决定哪些模块可以被外部导入。

```
__all__ = ["module_name1", "module_name2"]
```

B、批量导入

如果 Package 内多个模块文件都需要用到某些模块文件时，可以在 Package 的__init__.py 文件中导入需要的模块文件，然后就可以在

本 Package 不同的模块文件中直接使用导入的模块 。

（3）Module

Python 中一个 py 文件就是一个模块，Python 使用 import 或者 from...import 来导入相应的模块，包和模块不会被重复导入。

将整个模块(module)导入，格式为： import module

从某个模块中导入某个函数或变量，格式为： from module import function

从某个模块中导入多个函数，格式为： from module import firstfunc, secondfunc, thirdfunc

将某个模块中的全部函数或变量导入，格式为： from module import *

对导入的模块进行重命名

import package.module as name

import 导入包或模块时，如果导入模块较多需要换行，可以使用反斜杠，或使用小括号包含所有模块。

（4）module 内置变量

module 内部预定义了内置变量：

__name__：当前模块名称

__package__：当前模块所属包的名称

__doc__：当前模块的注释内容

__file__：当前文件

```
# -*- coding:utf-8 -*-
print("name: " + __name__)
```

```
print("package: " + (__package__ or "当前模块不属于任何包"))
print("doc: " + (__doc__ or "当前模块没有任何注释"))
print("file: " + __file__)
# name: __main__
# package: 当前模块不属于任何包
# doc: 当前模块没有任何注释
# file: test.py
```

如果某个文件被当作入口文件，内置变量__package__没有值，__name__值为__main__，__file__值为文件名称（不含路径）。

每个模块都可以任意写一些没有缩进的代码，并且在载入时自动执行，为了区分模块是主执行文件还是被调用的模块文件，Python 引入了一个变量__name__，当文件是被调用时，__name__的值为模块名，当文件作为入口被执行时，__name__为'__main__'。因此，可以在每个模块中写上测试代码，测试代码仅当模块被 Python 直接执行时才会运行，代码和测试结合在一起，完美实现对测试驱动开发（TDD）的支持。

```
if __name__ == "__main__":
    function_name()
```

将一个模块文件作为模块执行而不是应用执行的命令如下：

python3 -m package.module

2、Package 管理

（1）导入包和模块

import 只能导入包和模块，不能直接导入变量或者函数。对于多层包嵌套后导致导入名称过长，可以为其重命名。

```
import package1.package2.module_name
import package1
import package1.module_name
import package1.package2.module_name as new_name
```

导入包和模块时需要避免循环导入，两个或者多个模块文件互相导入会报错。

Python 在导入模块时，会执行模块里的所有内容，但多次导入只会执行一次。

import 导入包或模块时，Python 解释器寻找模块的优先级如下：

A、当前目录

B、环境变量 PYTHONPATH

C、sys.path(list 类型)

模块被导入执行时，Python 解释器为加快程序启动速度，会在与模块文件同一目录下自动生成.pyc 文件，.pyc 是经过编译后的字节码。

Python 使用缩进对齐组织代码的执行，所有没有缩进的代码（非函数定义和类定义），都会在载入时自动执行。

（2）导入变量和函数

导入某个模块的变量，多个使用逗号分隔

```
from package1.package2.module_name import variable_name
```

导入某个模块的函数，多个使用逗号分隔

```
from package1.package2.module_name import function_name
```

导入某个包的某个模块，多个使用逗号分隔

```
from package1 import module_name
```

导入模块的所有变量和函数

```
from package1.module_name import *
```

在模块内使用内置__all__属性指定本模块可以导出的变量或函数，外部导入只能使用指定的变量或函数。

在模块(*.py)中使用导出__all__列表里的类、函数、变量等成员，否

则将导出所有不以下划线开头（私有）的成员，在模块中使用__all__属性可避免在相互引用时的命名冲突。

```
#!/usr/bin/python3
__all__ = ["variable1", "variable2", "function_name1", "function_name2"]
variable1 = 0
variable2 = 0

def function_name1(args):
    pass

def function_name2(args):
    pass
```

在导入变量和函数时，如果变量和函数太多需要换行，则可以使用反斜杠换行或是使用小括号修饰多个变量或函数。

```
from package1.module_name import variable1, variable2 \
    variable3, function_name1
from package1.module_name import (variable1, variable2,
                                  function_name2, function_name2)
```

（3）顶级包

顶级包与入口文件 main.py 的位置有关，与 main.py 同级的包是顶级包，因此 main.py 入口文件不属于任何包。

（4）相对导入

相对导入是导入模块时指定被导入模块名称的相对路径。import 不支持相对导入，只能使用 from … import …格式实现相对导入，一个点表示当前包，两个点表示上一级包，以此类推。

Python 入口文件中没有 Package 的概念，因此不能使用相对导入。使用相对导入不要超出顶级包，入口文件同级包不能使用相对导入。main.py 如果作为模块执行时，可以使用相对导入，此时使用如下命令执行：python3 -m main.py

（5）绝对导入

绝对导入是导入时必须指定从顶级包到被导入模块名称的完整路径，可以使用 import 和 from … import …进行导入。

四、Python函数

1、函数定义

Python 使用 def 关键字定义函数，格式如下：

```
def function_name(parameter_list):
    pass
```

默认情况下，参数值和参数名称按函数声明中定义的顺序进行匹配。

函数的第一行语句可以选择性地使用文档字符串描述函数说明。

函数使用 return［表达式］结束函数，返回一个或多个值给调用方。

不带表达式的 return 相当于返回 None。

2、函数调用

函数调用直接使用定义的函数名称以及传递相应的参数进行调用。

Python 默认设置了函数的最大递归调用次数，开发者可以直接进行设置。设置方法如下：

```
sys.setrecursionlimit(100) # 设置最大递归次数为100
```

3、函数参数

（1）必须参数

必须参数是函数调用时必须传递的参数。

（2）关键字参数

关键字参数用于函数调用时，使用形式参数关键字将实际参数赋值给函数的相应形式参数，函数调用过程中传递实际参数顺序可以与形式参数顺序不匹配，但所有必须参数必须被赋值。

```
def add(x, y):
    result = x + y
    return result

add(y=2, x=3)
```

（3）默认参数

函数在定义过程中可以为参数指定默认值，并且必须参数必须放在默认参数的左侧。

函数调用时，可以使用关键字参数对形式参数进行赋值，默认参数可以缺省，但必须参数必须被赋值，必须参数可以使用关键字参数赋值。如果必须参数不使用关键字参数，则必须参数必须严格按照函数定义的顺序传递，其默认参数可以使用关键字参数选择性赋值，没有使用关键字参数传递的默认参数将使用默认值。

（4）可变参数

可变参数通过使用*修饰形式参数进行定义，在函数调用时传递可变的多个实际参数。

对于定义了可变参数的函数，如果需要传入序列作为实际参数，需要在序列前加*。

如果函数定义时有可变参数和默认参数，函数调用时可以使用关键字参数用于指定默认参数。

Python 中，如果函数定义时可变参数放在默认参数的左侧，则默认参数需要使用关键字参数显式指定。

（5）关键字可变参数

关键字可变参数通过使用**修饰形式参数进行定义，为字典类型，属于可选参数。

函数调用时，关键字可变参数可以传递多个键值对，也可以直接传递 dictionary 类型变量，此时需要使用**修饰字典变量。

4、匿名函数

Python 使用 lambda 表达式来创建匿名函数，函数定义如下：

lambda [arg1 [,arg2,.....argn]]:expression

Lambda 表达式定义的匿名函数的函数体不是一个代码块，因此，仅能在 lambda 表达式中封装有限的逻辑。

Lambda 表达式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 函数只能写一行，却不等同于 C 或 C++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。

```
sum = lambda arg1, arg2: arg1 + arg2;
# 调用 sum 函数
print(sum(10, 20))
print(sum(20, 20))
```

5、变量作用域

定义在函数内部的变量拥有局部作用域，定义在函数外的拥有全局作用域，代码块内部局部变量会覆盖全局变量，在局部作用域内修改全局变量的值时，需要使用 global 关键字对相应全局变量进行声明，

表明本作用域内使用的是全局变量。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。

```
base = 100

def sum(*args):
    global base
    base = 0
    result = 0;
    for i in args:
        result += i
    return result + base;

c = sum(1,2,3,4,5,6,7,8,9,10)
print(c) # 55
print(base) # 0
```

6、迭代器

迭代器是一个可以记住遍历位置的对象，只能往前不会后退。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器有两个基本的方法：iter() 和 next()。

```
list=[1,2,3,4]
it = iter(list)    # 创建迭代器对象
for x in it:
    print (x, end=" ")

list = [1, 2, 3, 4]
it = iter(list)  # 创建迭代器对象
while 1:
    print(next(it))
```

创建自定义迭代器类需要实现两个方法__iter__()与__next__() 。

```
class NumbersIter:
    def __iter__(self):
        self.a = 1
```

```
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

numbersIter = NumbersIter()
myiter = iter(NumbersIter)
print(next(myiter))
```

7、generator

在 Python 中，生成器（generator）是一种返回一个值的迭代器，每次从迭代器取下一个值。

（1）生成器表达式

生成器表达式用圆括号来创建生成器，其语法与推导式相同，只是将[]换成()。生成器表达式会产生一个新的生成器对象，即隐式生成器。

```
print(type([i for i in range(5)])) # <class 'list'>
print(type((i for i in range(5)))) # <class 'generator'>
x = (i for i in range(5))
print(next(x)) # 0
print(next(x)) # 1
```

（2）生成器函数

Python 中使用 yield 的函数为生成器函数，即显式生成器。

在调用生成器运行过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值，并在下一次执行 next()方法时从当前位置继续运行。

```
def square(N):
    for x in range(N):
        yield x**2

for item in square(5):
    print(item, end=' ')
```

五、Python面向对象

1、类定义

Python 默认所有的成员都是公有成员，但以两个下划线开头的名字表示私有成员，私有成员不允许直接访问，只能通过内部方法进行访问，私有成员也不允许被继承。

Python 通过在类变量、实例变量、类方法、实例方法前加__前缀，可以将其对外进行隐藏，变为类的私有变量或函数。由于 Python 中内置变量或函数使用__前后缀，因此，不推荐私有的变量或函数加__前后缀，只加__前缀。

Python 作为动态语言，允许类或实例动态增加属性，与类内部的私有的属性并不相同。

Python 维护一个用于保存类实例对象的数据字典，字典内将私有成员改名为_ClassName + __variable_name，因此在类外通过实例访问私有变量新的名称可以访问相应的私有变量。

2、类属性

（1）类属性

类属性是类的所有实例共享的属性，直接定义在 class 下，因此默认情况下类属性值只会保留一份，而不会为类的每个实例都保存一份。

类属性可以使用 ClassName.VariableName 访问，在实例方法内部可以使用 self.__class__.VariableName 进行访问。

类属性可以通过类直接访问，也可以直接通过实例进行访问；如果通过类的某个实例对类属性进行修改，本质上是为实例添加了一个与类属性名称相同的实例属性，对真正的类属性没有影响，因此不会影响其它实例获取类属性的值；通过类对类属性进行修改，必然会改变类属性的值，对类的所有实例是都有影响的。

（2）实例属性

实例属性又称成员属性或成员变量，是类的每个实例对象单独持有的属性，必须在类的__init__构造方法中进行声明。

实例属性可以直接通过实例对象来访问和更改，是每个实例对象独有的，某个实例对象的实例属性被更改不会影响其它实例对象的相同属性的值。Python 作为动态语言，可以在类外部动态增加实例的属性。

（3）私有属性

私有属性和实例属性必须在__init__构造方法中进行声明，但私有属性的属性名需要以双下划线__开头，比如 Person 中__id 属性。私有属性是一种特殊的实例属性，只允许在实例对象的内部（成员方法或私有方法中）访问，而不允许在实例对象的外部通过实例对象或类来直接访问，也不能被子类继承。

3、类方法

（1）成员方法

成员方法通过类的实例对象去访问，第一个参数必须是当前实例，通常写为 self；但也可以通过类名来调用成员方法，此时需要手动的传

递一个类的实例对象给成员方法的 self 参数。

（2）私有方法

私有方法是以双下划线开头的成员方法，只能在实例方法内部访问，且不能被子类继承；私有方法的第一个参数必须是当前实例对象本身，通常写为 self。通常，前后加双下划线的命名方式用于 Python 内置的方法，不推荐自定义方法使用。如果开发者以前后加双下划线的方式命名成员方法，则相应成员方法是公有的。

（3）类方法

类方法是以@classmethod 来修饰的成员方法，类方法要求第一个参数必须是当前类。类方法可通过实例对象进行访问，还可以直接通过类名进行访问，且第一个参数表示当前类，通常写为 cls。类方法只能访问类属性，不能访问实例属性，因此第一个参数传递代表当前类的 cls，而不是表示实例对象的 self。

（4）静态方法

静态方法是以@staticmethod 来装饰的成员方法，通常通过类名进行访问，也可以通过类的实例对象进行访问。本质上，静态方法已经与类没有任何关联，因为静态方法不要求必须传递实例或类参数。

静态方法内部可以访问类变量，可以直接使用 ClassName.Varaible_Name 方式对类变量进行访问。

静态方法对参数没有要求，因此可以任意给静态方法定义参数，如果给静态方法定义表示当前类的参数，那么就可以访问类属性；如果给静态方法定义了表示当前类的实例对象的参数，那么就可以访问实例

属性；如果没有给静态方法定义当前类参数或当前实例参数，那么就不能访问类或实例对象的任何属性。

（5）属性方法

属性方法是以@property 来装饰的成员方法，是以访问实例属性的方式对实例属性进行访问的成员方法；属性方法第一个参数必须是当前实例对象，且属性方法必须要有返回值。

Python 中属性方法通常用于在属性方法内部进行一系列的逻辑计算，最终将计算结果返回。

（6）方法绑定

类内部定义的方法，在没有被任何装饰器修饰时，通常绑定到类的实例，self 关键字含有自动传值过程。默认情况下，在类内部定义的方法都是绑定到对象的方法。绑定方法绑定到谁的身上，谁就作为第一个参数进行传入，绑定到类的方法给对象使用是没有任何意义的。

绑定到对象的方法，调用时会将对象参数自动传入；绑定到类的方法，调用时会将类作为参数自动传入。

静态方法是非绑定方法，不与类或对象绑定，谁都可以调用，没有自动传值效果。非绑定方法不与类或对象绑定，类和对象都可以调用，但没有自动传值。

4、类继承

（1）派生类定义

Python 中类继承按照父类中的方法是否已实现可分为两种：

A、实现继承：指直接继承父类的属性和已定义并实现的的方法；

B、接口继承：仅继承父类的属性和方法名称，子类必须自行实现方法的具体功能代码。

（2）派生类构造函数

派生类构造函数需要显式调用父类构造函数，对父类的属性成员进行初始化，调用父类构造函数时需要显式传递实例对象 self。

子类需要在自己的__init__方法中的第一行位置调用父类构造方法，上述代码给出了两种方法：

super(子类名，self).__init__(父类构造参数)，如 super.(Teacher, self).__init__(name，age)，推荐方式。

父类名.__init__(self，父类构造参数)，如 Person.__init__(self，name，age)。

（3）isinstance

isinstance 可以判断变量是否是某一种数据类型，也可以判断对象是否是类对象或者是类的子类对象。

issubclass 用来判断一个类是否是某个类的子类，返回的是一个 bool 类型数据。

5、多继承

（1）多继承

Python 支持多层父类继承，子类会继承父类所有属性和方法，包括父类的所有属性和方法。Python 虽然支持多继承，但对多继承的支

持有限。

多继承时，使用 super 只会调用第一个父类的属性方法，因此，要想调用特定父类的构造函数只能显式调用父类名.__init__。

如果父类中有相同的方法名，而在子类使用时未显式指定调用的具体父类方法，Python 会根据继承顺序从左至右搜索查找父类中是否包含方法。

如果子类从多个父类派生，而子类没有自己的构造函数时，按顺序继承，哪个父类在最前面且有自己的构造函数，就继承其构造函数。

如果子类从多个父类派生，而子类没有自己的构造函数时，如果最前面第一个父类没有构造函数，则依次查找后序继承父类的构造函数。

（2）多继承查找顺序

类的属性__mro__或者方法 mro()能打印出类的继承顺序，super()在执行时查找 MRO 列表,到列表当前位置的类中去查找其下一个类。为了实现继承，Python 会在 MRO 列表上从左到右开始查找基类，直到找到第一个匹配属性的类为止。

Python 3.x 中无论是否显式指定继承 object，所有类都是新式类，在多继承的情况下，经典类查找父类属性或方法的顺序是深度优先，新式类查找父类属性的顺序是广度优先。

super 是 MRO 中的一个类。MRO 全称 Method Resolution Order，代表类的继承顺序。对于定义的每一个类，Python 会计算出一个方法解析顺序(MRO)列表，MRO 列表是一个简单的所有基类的线性顺序列表。

MRO 列表的构造是通过一个 C3 线性化算法来实现的，MRO 会合并所有父类的 MRO 列表并遵循如下三条准则:

A、子类会先于父类被检查。

B、多个父类会根据它们在列表中的顺序被检查。

C、如果对下一个类存在两个合法的选择，选择第一个父类。

MRO 可以保证多继承情况每个类只出现一次，super().init 相对于类名.init，在单继承上用法基本无差，但在多继承上，super 方法能保证每个父类的方法只会执行一次，而使用类名的方法会导致方法被执行多次。

单继承时，使用 super 方法，不能全部传递，只能传父类方法所需的参数，否则会报错；Python3 中多继承时，使用类名.init 方法需要把每个父类全部写一遍，而使用 super 方法只需一条语句便执行全部父类的方法，因此多继承需要全部传参。

6、多态机制

多态通过继承接口方式实现，Python 中没有接口，但 Python 可以通过在成员方法中抛出一个 NotImplementedError 异常来强制继承接口的子类在调用接口方法前必须先实现接口方法。

```
class Animal(object):  # Animal Interface
    def __init__(self, name):
        self.name = name

    def walk(self):
        raise NotImplemented('Subclass must implement the abstract method')

    def talk(self):
        raise NotImplemented('Subclass must implement the abstract method by se
```

```
lf')


class Dog(Animal):
    def talk(self):
        print('%s is talking: wang wang...' % self.name)

    def walk(self):
        print('%s is a Dog, walk by 4 legs' % self.name)


class Duck(Animal):
    def talk(self):
        print('%s is talking: ga ga...' % self.name)

    def walk(self):
        print('%s is a Duck, walk by 2 legs' % self.name)


if __name__ == "__main__":
    dog = Dog('Trump')
    dog.talk()
    dog.walk()

    duck = Duck('Tang')
    duck.talk()
    duck.walk()
```

接口的所有子类必须实现接口中定义的所有方法；接口的各个子类在实现接口中同一个方法时，具体的代码实现各不相同，即多态。

7、反射机制

Python 反射机制通过 hasattr、getattr、setattr、delattr 四个内置函数实现，内置函数不仅可以用在类和对象中，也可以用在模块等。hasattr(key)返回一个 bool 值，判断某个成员或者属性在不在类或者对象中。

getattr(key,default=xxx)获取类或者对象的成员或属性，如果不

存在，则会抛出 AttributeError 异常，如果定义了 default 那么当没有属性时会返回默认值。

setattr(key,value)假如有 key 属性，那么更新 key 属性，如果没有就添加 key 属性并赋值 value。

delattr(key)删除某个属性。

8、单例模式

在面向对象编程中，单例模式是一个类只有一个对象，所有的操作都通过单例对象来完成，实现代码如下：

```
class Instance:
    __instance = None

    @classmethod
    def get_instance(cls):
        if cls.__instance:
            return cls.__instance
        else:
            cls.__instance = Instance
            return cls.__instance

obj1 = Instance.get_instance()
print(id(obj1))
obj2 = Instance.get_instance()
print(id(obj2))
```

六、Python并发编程

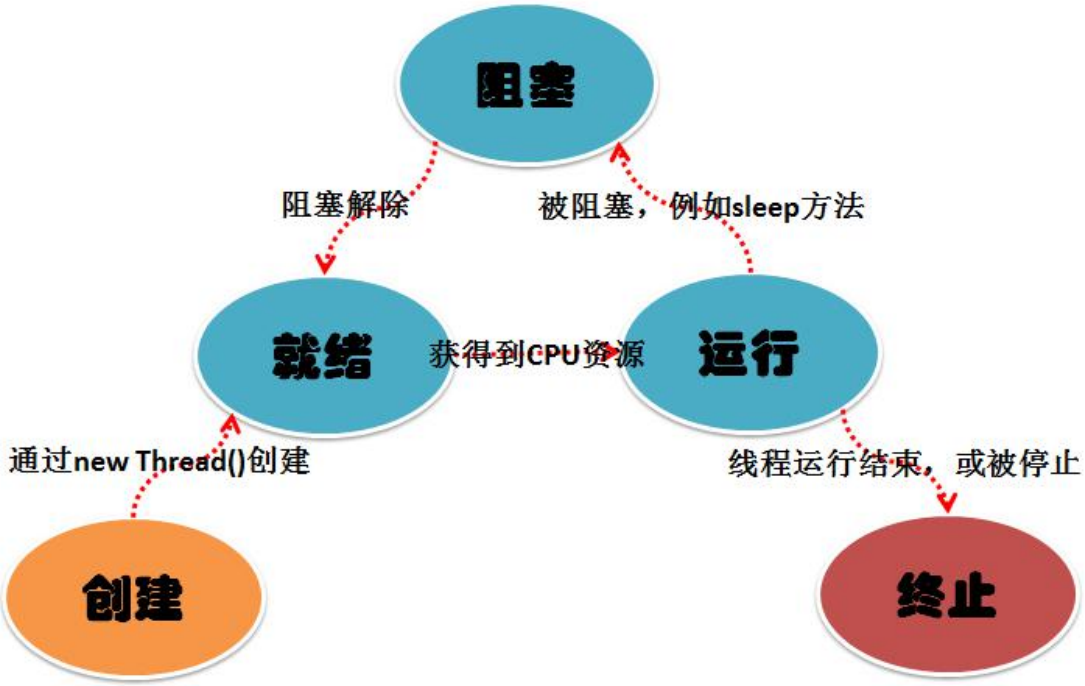
1、Python 线程模块

线程有就绪、阻塞、运行三种基本状态。

就绪状态是指线程具备运行的所有条件，在等待 CPU 执行；

运行状态是指线程占有 CPU 正在运行；

阻塞状态是指线程在等待一个事件，逻辑上不可执行。



Python3 通过 threading 模块提供对线程的支持。

multiprocessing 模块是跨平台版本的多进程模块，提供 Process 类代表进程对象。创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 Process 实例。terminate 强制终止进程，不会进行任何清理操作，子进程会变成僵尸进程，不会释放进程锁导致死锁。

2、全局解释锁 GIL

Python 虽然提供了多线程模块，但如果想通过多线程提高代码速度，并不推荐使用多线程模块。Python 全局锁 GIL 会确保任何时候多个线程中只有一个会被执行，即轮流执行。

全局锁 GIL 并不是 Python 特性，而是在实现 Python 解析器(CPyt

hon)时所引入概念，Python 完全可以不依赖于 GIL。Python 有 CPython、PyPy、Psyco 等不同 Python 解释器，JPython 没有 GIL，CPython 是大部分环境下默认的 Python 执行环境。

GIL 限制同一时刻只能有一个线程运行，无法发挥多核 CPU 优势。GIL 本质是互斥锁，将并发运行变成串行运行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。Python 进程内，不仅有主线程或者主线程开启的其它线程，还有 Python 解释器开启的垃圾回收等解释器级别线程。进程内，所有数据都是共享的，代码作为一种数据也会被所有线程共享，多个线程先访问到解释器的代码，即拿到执行权限，然后将 target 代码交给解释器代码去执行，解释器代码是所有线程共享的，所以垃圾回收线程也可能访问到解释器的代码而去执行，因此为了保证数据安全需要加锁处理，即 GIL。

由于 GIL 存在，同一时刻同一进程中只有一个线程被执行。多核 CPU 可以并行完成计算，因此多核可以提升计算性能，但 CPU 一旦遇到 IO 阻塞，仍然需要等待，所以多核 CPU 对 IO 密集型任务提升不明显。根据执行任务是计算密集型还是 IO 密集型，不同场景使用不同的方法，对于计算密集型任务，多进程占优势；对于 IO 密集型任务，多线程占优势。

3、线程创建

（1）threading.Thread 实例化

threading.Thread 构造函数如下：

```
def __init__(self, group=None, target=None, name=None,
             args=(), kwargs=None, *, daemon=None):
```

（2）threading.Thread 子线程

直接从 threading.Thread 派生新线程类，在子类中重写 run()和 init()方法，实例化后调用 start()方法启动新线程，start 函数内部会调用线程的 run()方法。

如果需要从外部传入函数，可以将传入参数作为子线程实例属性，在 run 实例方法内进行调用。

```
class WorkThread(threading.Thread):
    def __init__(self, target, args):
        threading.Thread.__init__(self)
        self.target = target
        self.args = args

    def run(self):
        print("start thread: ", self.name)
        self.target(*self.args)
        print("end thread: ", self.name)
```

当一个进程启动后，默认产生一个主线程，因为线程是程序执行流的最小单元，多线程时主线程会创建多个子线程。在 Python 中，默认情况下主线程执行完自己任务后就会退出，但子线程会继续执行自己任务，直到任务结束。

当使用 setDaemon(True)方法设置子线程为守护线程时，主线程一旦执行结束，则全部线程都会终止执行，不管子线程任务是否执行结束。设置 setDaemon 必须在启动子线程前进行设置。

join 方法用于让主线程等待子线执行完毕并返回结果后，再执行主线程剩余任务，子线程不执行完，主线程会一直等待。join 有一个 timeout 参数，当设置守护线程时，主线程对子线程等待 timeout 时

间，给每个子线程一个 timeout 时间，让子线程执行，时间一到，不管任务有没有完成，直接杀死。如果有多个子线程，全部等待时间是每个子线程 timeout 的累加和。

没有设置守护线程时，主线程将会等待 timeout 的累加和的一段时间，时间一到，主线程结束，但并不会杀死子线程，子线程依然可以继续执行，直到子线程全部结束，程序退出。

4、线程同步

如果多个线程共同对某个数据修改，为保证数据的正确性，需要对多个线程进行同步。

（1）互斥锁

threading.Thread 类 Lock 锁和 Rlock 锁可以实现简单线程同步，Lock 锁和 Rlock 锁都有 acquire 方法和 release 方法。

（2）信号量

互斥锁同时只允许一个线程访问共享数据，而信号量同时允许一定数量的线程访问共享数据。

（3）条件变量

条件变量能让一个线程 A 停下来，等待其它线程 B，线程 B 满足了某个条件后通知（notify）线程 A 继续运行。线程首先获取一个条件变量锁，如果条件不满足，则线程等待（wait）并释放条件变量锁；如果条件满足则执行线程，也可以通知其它状态为 wait 的线程。其它处于 wait 状态的线程接到通知后会重新判断条件。

（4）事件

事件用于线程间通信，线程发出一个信号，其它一个或多个线程等待，调用 event 对象的 wait 方法，线程则会阻塞等待，直到其它线程 set 后，才会被唤醒。

（5）线程优先级队列

Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括 FIFO（先入先出)队列 Queue，LIFO（后入先出）队列 LifoQueue，优先级队列 PriorityQueue，能够在多线程中直接使用，可以使用队列来实现线程间同步。

（6）线程死锁

死锁是指两个或两个以上的进程或线程在执行过程中，因争夺资源而造成的一种互相等待的现象。在线程间共享多个资源的时候，如果分别占有一部分资源并且同时在等待对方的资源，就会造成死锁。

解决死锁问题的一种方案是为程序中的每一个锁分配一个唯一的 id，然后只允许按照升序规则来使用多个锁。

5、线程池

（1）线程池简介

concurrent.futures 模块中的 Executor 是线程池的抽象基类，Executor 提供了 ThreadPoolExecutor 和 ProcessPoolExecutor，ThreadPoolExecutor 用于创建线程池，ProcessPoolExecutor 用于创建进程池。

（2）线程池

ThreadPoolExecutor(max_works)，如果未显式指定 max_works，默认线程池会创建 CPU 的数目*5 数量的线程 。

```
# -*- coding:utf-8 -*-
from concurrent.futures import ThreadPoolExecutor
import threading
import time
import os
import string


class WorkThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        print('Process[%s]:%s start and run task' % (os.getpid(), threading.cur
rentThread().getName()))
        time.sleep(2)
        return "Process[{}]:{} end".format(os.getpid(), threading.currentThrea
d().getName())


def work_task(thread_name):
    print('Process[%s]:%s start and run task' % (os.getpid(), threading.curren
tThread().getName()))
    time.sleep(5)
    return "Process[{}]:{} end".format(os.getpid(), threading.currentThread().
getName())


def get_call_back(future):
    print(future.result())


if __name__ == '__main__':
    print('main thread start')

    # create thread pool
    thread_pool = ThreadPoolExecutor(5)
    futures = []
```

```
    for i in range(5):
        thread = WorkThread()
        future = thread_pool.submit(thread.run)
        futures.append(future)

    for i in range(5):
        future = thread_pool.submit(work_task, i)
        futures.append(future)

    for future in futures:
        future.add_done_callback(get_call_back)

    # thread_pool.map(work_task, (2, 3, 4))
    thread_pool.shutdown()
```

（3）进程池

ProcessPoolExecutor(max_works)，如果未显式指定 max_works，默认进程池会创建 CPU 的数目*5 数量的进程 。

进程池同步方案：

```
from concurrent.futures import ProcessPoolExecutor
import os
import time
import random


def work_task(n):
    print('Process[%s] is running' % os.getpid())
    time.sleep(random.randint(1,3))
    return n**2


if __name__ == '__main__':
    start = time.time()
    pool = ProcessPoolExecutor()
    for i in range(5):
        obj = pool.submit(work_task, i).result()
    pool.shutdown()
    print('='*30)
    print("time: ", time.time() - start)
```

进程池异步方案：

```
from concurrent.futures import ProcessPoolExecutor
import os
import time
import random


def work_task(n):
    print('Process[%s] is running' % os.getpid())
    time.sleep(random.randint(1, 3))
    return n**2


if __name__ == '__main__':
    start = time.time()
    pool = ProcessPoolExecutor()
    objs = []
    for i in range(5):
        obj = pool.submit(work_task, i)
        objs.append(obj)
    pool.shutdown()
    print('='*30)
    print([obj.result() for obj in objs])
    print("time: ", time.time() - start)
```

七、Python装饰器

1、闭包简介

（1）嵌套函数

如果一个函数内部定义了另一个函数，则是嵌套函数。外部函数即外函数，内部的函数即内函数。嵌套函数示例如下：

```
def outer():
    x = 1
    def inner():
        y = x + 1
        print(y)
    inner()
```

outer 函数内定义了一个 inner 函数，并调用。inner 函数在自己作

用域内查找局部变量失败后，会进一步向上一层作用域里查找。如果 outer 函数里不直接调用 inner 函数，而通过 return inner 返回一个 inner 函数的引用，将会得到一个 inner 函数对象，而非运行结果。

```
def outer(x):
    a = x
    def inner(y):
        b = y
        print(a + b)
    return inner

if __name__ == '__main__':
    func = outer(1)
    func(10)
```

如果将外函数的变量 x 换成被装饰函数对象(func)，内函数的变量 y 换成被装饰函数的参数，可以得到一个通用的装饰器。

```
def decorator(func):
    def inner(*args, **kwargs):
        # add_other_actions()
        return func(*args, **kwargs)
    return inner
```

（2）闭包

如果外函数中定义了一个内函数，且内函数体内引用到了体外的变量，外函数通过 return 返回内函数的引用时，会把定义时涉及到的外部引用变量和内函数打包成一个整体（闭包）返回。

```
def outer(x):
    a = x
    def inner(y):
        b = y
        print(a + b)
    return inner

if __name__ == '__main__':
    func = outer(1)
    func(10)
```

通常，函数运行结束时，临时变量会被销毁。但闭包中，当外函数发

现自己的临时变量会在将来的内函数中用到，外函数在结束时，返回内函数同时，会把外函数的临时变量同内函数绑定在一起。即使外函数已经结束，内函数仍然能够使用外函数的临时变量。

2、装饰器简介

（1）Python 装饰器简介

Python 装饰器是用于拓展原来函数功能的一种函数，可以在不改变一个函数代码和调用方式的情况下给函数添加新的功能

装饰器本质是一个闭包，把函数当做参数然后返回一个替代版函数，是对函数或类功能的低耦合功能增强。

（2）Python 装饰器的特点

开放封闭原则，即对扩展开放，对修改封闭；

装饰器本质可以是任意可调用的对象，被装饰的对象也可以是任意可调用对象；

装饰器的功能是在不修改被装饰器对象源代码以及被装饰器对象的调用方式的前提下为其扩展新功能；

（3）装饰器编写

```
import time

def elapsed(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print("elapsed time: {} ms".format((end - start) * 1000))
    return wrapper

@elapsed
```

```
def hello():
    print("Hello!")

if __name__ == '__main__':
    hello()
```

被装饰器装饰的函数的名字已经改变为装饰器的内函数名字，hello.__name__值为 wrapper，因为外函数返回由 wrapper 函数和其外部引用变量组成的闭包。

为了保证装饰过的函数__name__属性不变，可以使用 functools 模块里的 wraps 方法，先对 func 变量进行 wraps。

```
import time
from functools import wraps

def elapsed(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print("function {} elapsed time: {} ms".format(func.__name__, (end - st
art) * 1000))
    return wrapper

@elapsed
def hello():
    print("Hello!")

if __name__ == '__main__':
    hello()
```

八、Python性能优化

1、Python 性能分析工具

（1）timeit

```
def timeit(stmt="pass", setup="pass", timer=default_timer,
           number=default_number):
```

```
    return Timer(stmt, setup, timer).timeit(number)

def repeat(stmt="pass", setup="pass", timer=default_timer,
           repeat=default_repeat, number=default_number):
    return Timer(stmt, setup, timer).repeat(repeat, number)
```

timeit 模块的 timeit 和 repeat 方法本质是定义 Timer 类，stmt 参数是要计时的语句或者函数，setup 参数是为执行语句构建环境的导入语句。

```
import timeit
code = """
sum = []
for i in range(1000):
    sum.append(i)
"""
print(timeit.timeit(stmt="[i for i in range(1000)]", number=100000))
print(timeit.timeit(stmt=code, number=100000))
print(timeit.repeat(stmt="[i for i in range(1000)]", repeat=2, number=10000
0))
```

（2）time 装饰器

```
def execute_time(func):
    from time import time
    # 定义嵌套函数，用来打印出装饰的函数的执行时间
    def wrapper(*args, **kwargs):
        # 定义开始时间
        start = time()
        # 执行函数
        func_return = func(*args, **kwargs)
        # 定义结束时间
        end = time()
        # 打印方法名称和其执行时间
        print('{}() execute time: {}s'.format(func.__name__, end - start))
        # 返回 func 的返回值
        return func_return
    # 返回嵌套的函数
    return wrapper
```

（3）cProfile

cProfile 使用方式：

python -m cProfile filename.py -o report

2、Python 性能优化建议

（1）优化算法时间复杂度

算法的时间复杂度对程序的执行效率影响最大，在 Python 中可以通过选择合适的数据结构来优化时间复杂度，如 list 和 set 查找某一个元素的时间复杂度分别是O(n)和O(1)。不同场景有不同的优化方式，一般有分治、分支界限、贪心、动态规划等思想。

（2）合理使用 copy 和 deepcopy

对于 dict 和 list 等数据结构对象，直接赋值使用引用方式。对于需要复制整个对象，使用 copy 和 deepcopy，deepcopy 是深拷贝。

（3）使用 dict 或 set 查找元素

Python 中 dict 和 set 使用 hash 表实现，查找元素的时间复杂度是O(1)。

（4）合理使用生成器 generator

使用()得到 generator 对象，所需要的内存空间与列表的大小无关，所以效率会高一些。比如 set(i for i in range(100000))会比 set([i for i in range(100000)])快。

后者的效率反而更高，但如果循环里有 break，用 generator 的好处是显而易见的。yield 也是用于创建 generator

（5）优化循环

循环外能做的事不要放在循环内，避免每次循环迭代都进行计算。

（6）优化表达式中多个判断表达式的顺序

对于 and，应该把满足条件少的放在前面，对于 or，把满足条件多

的放在前面。

（7）使用 join 合并迭代器中字符串

join 累加字符串可以大幅提升性能。

（8）不借助中间变量交换两个变量的值

使用 a,b=b,a 而不是 c=a;a=b;b=c;交换 a,b 的值可大幅提高性能。

（9）使用 if is

使用 if is True 比 if == True 将近快一倍。

（10）while 1 比 while True 更快

while 1 比 while true 快很多，在 python2.x 中，True 是一个全局变量，而非关键字。

（11）使用幂运算符**替代 pow 函数

（12）使用 C 语言实现功能模块

使用 cProfile、cStringIO 和 cPickle 等用 C 语言实现相同功能（分别对应 profile、StringIO、pickle）的包。

（13）使用性能最佳序列化方式

json 比 cPickle 快近 3 倍，比 eval 快 20 多倍。

（14）使用 C 扩展模块

C 扩展模块主要有 CPython 原生 API、ctypes、Cython、cffi 四种方式，使得 Python 程序可以调用由 C 语言动态链接库。

CPython 原生 API：通过引入 Python.h 头文件，对应 C 程序中可以直接使用 Python 数据结构，实现过程相对繁琐，但适用广泛。

ctypes：通常用于封装(wrap)C 程序，让纯 Python 程序调用 C 语言动态链接库函数。如果想要在 Python 中使用已经有 C 类库，使用 ctypes 是很好的选择。

Cython：Cython 是 CPython 的超集，用于简化编写 C 扩展过程，Cython 优点是语法简洁，可以很好地兼容 numpy 等包含大量 C 扩展的库。Cython 使用场景针对项目中某个算法或过程的优化。

cffi：cffi 是 ctypes 在 PyPy 中的实现，兼容 CPython。cffi 提供在 Python 使用 C 类库的方式，可以直接在 Python 代码中编写 C 代码，同时支持链接到已有 C 类库。

（15）并行编程

因为 GIL 全局锁存在，Python 很难充分利用多核 CPU 的优势。但可以通过内置模块 multiprocessing 实现并行模式。

多进程：对于 CPU 密集型程序，可以使用 multiprocessing 的 Process、Pool 等封装类，通过多进程的方式实现并行计算。但因为进程间通信成本比较大，对于进程间需要大量数据交互的程序效率未必有大的提高。

多线程：对于 IO 密集型的程序，multiprocessing.dummy 模块使用 multiprocessing 的接口封装 threading，使得多线程编程变得非常简单(比如可以使用 Pool 的 map 接口，简洁高效)。

分布式：multiprocessing 中的 Managers 类提供可以在不同进程间共享数据的方式，可以开发出分布式程序。

（16）使用 PyPy 解释器

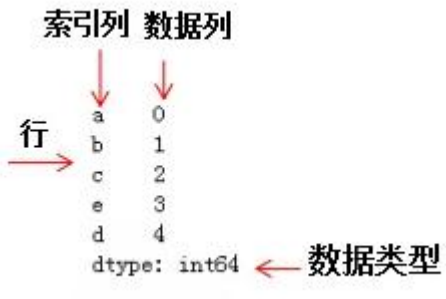
PyPy 是用 RPython(CPython 子集)实现的 Python 解释器，使用 Just-in-Time(JIT)编译器，利用程序运行过程的数据进行优化。根据官网基准测试数据，PyPy 比 CPython 实现的 Python 要快 6 倍以上。由于历史原 PyPy 目前还保留着 GIL，正在进行的 STM 项目试图将 PyPy 变成没有 GIL 的 Python。如果 Python 程序中含有 C 扩展(非 cffi 方式)，JIT 优化效果会大打折扣，甚至比 CPython 慢（比 Numpy），所以在 PyPy 中最好用纯 Python 或使用 cffi 扩展。

九、Pandas

1、Series

Series 是能够保存任何类型数据(整数，字符串，浮点数，Python 对象等)的一维标记数组，轴标签统称为 index（索引）。

series 是一种一维数据结构，每一个元素都带有一个索引，其中索引可以为数字或字符串。



pandas.Series(data, index, dtype, copy)

data：构建 Series 数据，可以是 ndarray、list、tuple、dict、constants。

index：索引值必须是唯一的和散列的，与数据长度相同。如果没有指定索引，默认为 np.arange(n)。

dtype：数据类型，如果没有，将推断数据类型。

copy：是否复制数据，默认为 false。

（1）ndarray 构建 Series

使用 ndarray 作为数据时，传递的索引必须与 ndarray 具有相同的长度。如果没有传递索引值，那么默认索引是 range(n)，其中 n 是数组长度，即[0,1,2,3.... range(len(array))-1] - 1]。

```
data = numpy.array([0, 1, 2, 3, 4])
s1 = pandas.Series(data, index=['a', 'b', 'c', 'd', 'e'])
s1 = pandas.Series(data)
```

（2）dict 构建 Series

使用字典(dict)作为数据时，如果没有指定索引，则按排序顺序取得字典键以构造索引。如果传递索引，索引中与标签对应的数据中的值将被取出。传递索引时，索引顺序保持不变，缺少的元素使用 NaN(不是数字)填充。

```
data = {'a': 1, 'b': 'hello', 'c': 'hello world', "d": "world"}
s2 = pandas.Series(data, index=['a', 'b', "hello", 'd'])
s2 = pandas.Series(data)
```

（3）常量值构建 Series

使用标量值作为数据，必须提供索引，会重复标量值以匹配索引长度。

```
s3 = pandas.Series(100, index=[1, 2, 3])
```

（4）序列构建 Series

使用 list、tuple 作为数据时，传递索引必须与 list、tuple 具有相同长度。如果没有传递索引值，那么默认索引是 range(n)，其中 n 是 list 的长度，即[0,1,2,3.... range(len(list))-1] - 1]。

```
s4 = pandas.Series((1, 2, 3, "hello"))
s4 = pandas.Series([1, 2, 3, "hello"])
```

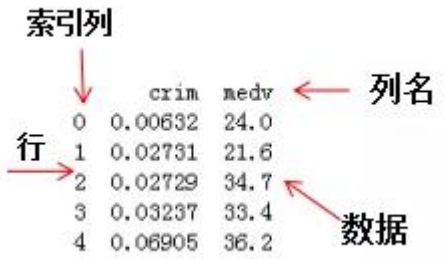
（5）Series 元素访问

Series 可以通过索引标签获取和设置值，使用索引标签值检索单个元素，使用索引标签值列表检索多个元素。如果使用不包含在索引内的标签，则会出现异常。

```
s = pandas.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
s['a'] = 101
```

2、DataFrame

DataFrame 是二维的表格型数据结构，即数据以行和列的表格方式排列，DataFrame 是 Series 的容器。



pandas.DataFrame(data, index, columns, dtype, copy)

data：构建 DataFrame 的数据，可以是 ndarray、series、map、lists、dict、constant 和其它 DataFrame。

index：行索引标签，如果没有传递索引值，索引默认为 numpy.arrange(n)。

columns：列索引标签，如果没有传递索列引值，默认列索引是 numpy.arange(n)。

dtype：每列的数据类型。

copy：用于复制数据，默认值为 False

（1）list 创建 DataFrame

使用单个列表或嵌套列表作为数据创建 DataFrame 时，如果不指定 index 或 columns，默认使用 range(len(list))作为 index，对于单列表，默认 columns=[0]，对于嵌套列表，默认 columns 为内层列表的长度的 range。

指定 index 或 columns 时，index 的长度必须与 list 长度匹配，columns 的长度必须与 list 的内层列表长度匹配，否则将报错。

```
data = [1, 2, 3, 4, 5]
df = pandas.DataFrame(data)
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
df = pandas.DataFrame(data)
df = pandas.DataFrame(data, index=['a', 'b', 'c'], columns=['A', 'D', 'C'])
```

（2）使用 ndarray 和 list 的字典创建 DataFrame

使用 ndarray、list 组成的字典作为数据创建 DataFrame 时，所有的 ndarray、list 必须具有相同的长度。如果传递 index，则 index 长度必须等于 ndarray、list 的长度，columns 为字典的 key 组成的集合。如果没有传递 index，则默认情况下，index 将为 range(n)，其中 n 为 list 或 ndarray 长度。

```
data = {'Name': ['Tom', 'Jack', 'Steve', 'Ricky'], 'Age': [28, 34, 29, 42]}
df = pandas.DataFrame(data)
df = pandas.DataFrame(data, index=['rank1', 'rank2', 'rank3', 'rank4'])
```

（3）使用字典列表创建 DataFrame

使用字典的列表作为数据创建 DataFrame 时，默认使用 range（len（list））作为 index，字典的 Key 集合作为 columns，如果字典没有相应键值对，其值使用 NaN 填充。当指定 columns 时，如

果 columns 使用字典的键集合以外元素作为 columns 的元素，则使用 NaN 进行填充，并提取出 columns 指定的数据源字典中相应的键值对。

```
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pandas.DataFrame(data)
df = pandas.DataFrame(data, index=['first', 'second'], columns=['a', 'b', 'C
'])
```

（4）使用 Series 字典创建 DataFrame

使用 Series 的字典作为数据创建 DataFrame 时，默认使用所有 Series 的 index 的并集作为 DataFrame 的 index，字典的 Key 集合作为 columns。当指定 columns 时，如果 columns 使用字典的 Key 集合以外元素作为 columns 的元素，则使用 NaN 进行填充，并提取出 columns 指定的数据源字典中相应的键值对。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
df = pandas.DataFrame(data, index=['a', 'b', 'c', 'A'], columns=['one', 'two',
 'three'])
```

（5）列操作

通过字典键可以进行列选择，获取 DataFrame 中的一列数据；通过向 DataFrame 增加相应的 Key 和 Series 值，可以为 DataFrame 增加一列。

通过 del 可以删除 DataFrame 的列。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
df['three'] = df['one'] + df['two']
del df['three']
```

（6）行操作

DataFrame 行选择可以通过将行标签传递给 loc 函数来选择行，也可以通过将整数位置传递给 iloc()函数来选择行，返回 Series，Series 的名称是检索的标签，Series 的 index 为 DataFrame 的 columns。DataFrame 多行选择可以通过使用:运算符对 DataFrame 进行行切片操作，选择多行。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
       'three': pandas.Series([10, 20, 30], index=['a', 'b', 'c'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
print(df.loc['a'])
print(df.iloc[0])
print(df[2:4])
```

DataFrame 的行追加通过 append 函数实现。

DataFrame 的行删除通过将索引标签传递给 drop 函数进行行删除，如果标签重复，则会删除多行。

3、DataFrame 数据选择

（1）通过标签获取行数据

Pandas 提供各种方法来完成基于标签的索引，可以使用标签如下：

A、单个标量标签

B、标签列表

C、切片对象，标签为切片时包括起始边界

D、一个布尔数组

loc 需要两个标签，用","分隔。第一个表示行，第二个表示列。

标签的优点是可以多轴交叉选择，可以通过行 index 标签和列标签定位 DataFrame 数据，但切片包含闭区间。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
       'three': pandas.Series([10, 20, 30], index=['a', 'b', 'c'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
print(df.loc['a', 'one'])
print(df.loc[:, 'one'])
print(df.loc['a', :])
print(df.loc['a':'c', :])
print(df.loc['a':'c', ['one', 'two']])
```

（2）通过位置获取行数据

Pandas 提供获取整数索引的多种方法，如整数、整数列表、Series 值。通过传递位置索引进行位置选择，位置索引可以使用切片操作。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
       'three': pandas.Series([10, 20, 30], index=['a', 'b', 'c'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
print(df.iloc[0:2, :])
print(df.iloc[0:2, [1, 2]])
print(df.iloc[0, 2])
```

（3）直接获取数据

DataFrame 可以通过列名获取整列的数据，通过切片获取多行数据。

```
data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
       'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
       'three': pandas.Series([10, 20, 30], index=['a', 'b', 'c'])}
df = pandas.DataFrame(data, columns=['one', 'two', 'three'])
print(df['one'])
print(df.one)
print(df[0:2])
print(df['c':'d'])
```

（4）通过 bool 索引获取数据

DataFrame 可以使用一个单独列的值来选择符合条件的行，也可以对 DataFrame 所有的值来选择数据，不满足条件的值填充 NaN。

```
df = pandas.DataFrame(numpy.random.randn(6, 3), columns=['A', 'B', 'C'])
print(df[df.A > 0])
print(df[df > 0])
```

4、函数应用

如果要将自定义函数或其它库函数应用于 Pandas 对象，有三种使用方式。pipe()将函数用于表格，apply()将函数用于行或列，apply map()将函数用于元素。

（1）表格函数应用

可以通过将函数对象和参数作为 pipe 函数的参数来执行自定义操作，会对整个 DataFrame 执行操作。

```
        def adder(x, y):
   return x + y
df = pandas.DataFrame(numpy.random.randn(5, 3),columns=['A', 'B', 'C'])
df = df.pipe(adder, 1)
```

（2）行列函数应用

使用 apply()函数可以沿 DataFrame 或 Panel 的轴执行应用函数，采用可选 axis 参数。 默认情况下，操作按列执行。

```
df = pandas.DataFrame(numpy.random.randn(5, 3),columns=['A', 'B', 'C'])
# 列操作，对列求和
result = df.apply(numpy.sum)
# 行操作，对行求和
result = df.apply(numpy.sum, axis=1)
```

（3）元素函数应用

在 DataFrame 的 applymap()函数可以接受任何 Python 函数，并且返回单个值。

```
df = pandas.DataFrame(numpy.random.randn(5, 3),columns=['A', 'B', 'C'])
result = df.applymap(lambda x: x**2)
```

5、数据清洗

（1）缺失值处理

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
', 'C', 'D'])
# 查看每一列的缺失值
print(df.isnull().sum())
# 查看每一列完整数据的数量
print(df.shape[0] - df.isnull().sum())
# 删除列名称为 D 的列
del df['D']
# 删除第 2 列
df.drop(df.columns[2], axis=1, inplace=True)
# 删除 B 列
df.drop('B', axis=1, inplace=True)
# 删除前两行
df.drop(df.index[0:2], axis=0, inplace=True)


df.dropna(self, axis=0, how='any', thresh=None, subset=None,
          inplace=False)
```

axis 为轴，0 表示对行进行操作，1 表示对列进行操作。

how 为操作类型，’any’表示只要出现 NaN 的行或列都删除，’all’表示删除整行或整列都为 NaN 的行或列。

thresh：NaN 的阈值，达到 thresh 时删除。

```
df.fillna(self, value=None, method=None, axis=None, inplace=False,
          limit=None, downcast=None, **kwargs)
```

value:填充的值，可以为字典，字典的 key 为列名称。

inplace:表示是否对源数据进行修改，默认为 False。

fillna 默认会返回新对象，但也可以对现有对象进行就地修改。

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
', 'C', 'D'])
df.iloc[0, 3] = None
df.iloc[2, 2] = None
print(df)
# 删除列中包含 Nan 值的列
print(df.dropna(axis=1))
# 删除包含 Nan 的所有行或列
```

```
print(df.dropna(how='any'))
# 使用 value 替换所有 Nan 值
print(df.fillna(value=3.14))
# 使用键值对按列替换 Nan
print(df.fillna(value={"C": 3.14, "D": 0}))
# # 使用前一行的元素进行填充
print(df.fillna(method='pad'))
```

（2）行列处理

通过字典键可以进行行列选择，获取 DataFrame 中的一列数据。

生成 DataFrame 时指定 index 和 columns

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
', 'C', 'D'])
df.index = pandas.date_range("20210201", periods=6)
df.columns = ['D', 'C', 'B', 'A']
df = pandas.DataFrame(numpy.random.randn(6, 4), columns=['A', 'B', 'C', 'D'])
df['date'] = pandas.date_range("20210201", periods=6)
print(df.set_index('date'))
```

对某列进行单位转换示例如下：

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
', 'C', 'D'])
df['D'] = [10000, 34000, 60000, 34000, 56000, 80000]
for i in range(len(df['D'])):
    weight = float(df.iloc[i, 3]) / 10000
    df.iloc[i, 3] = '{}万'.format(weight)
```

（3）重复值删除

```
df.duplicated(self, subset=None, keep='first')
```

检查 DataFrame 是否有重复数据。

subset：子集，列标签或列标签的序列

keep：可选值为 first，last，False，first 表示保留第一个出现的值，

last 表示保留最后一个出现的值，False 表示保留所有的值。

```
df.drop_duplicates(self, subset=None, keep='first', inplace=False)
```

删除 DataFrame 的重复数据。

subset：子集，列标签或列标签的序列

keep：可选值为 first，last，False，first 表示保留第一个出现的值，last 表示保留最后一个出现的值，False 表示保留所有的值。

inplace：值为 True 表示修改源数据，为 False 表示不修改源数据

```
data = [['Alex', numpy.nan, 80], ['Bob', 25, 90], ['Bob', 25, 90]]
df = pandas.DataFrame(data, columns=['Name', 'Age', 'Score'])
# 使用 bool 过滤，取出重复的值
print(df[df.duplicated(keep=False)])
# 删除重复值，修改源数据
df.drop_duplicates(keep='last', inplace=True)
```

（4）异常值处理

异常值分为两种，一种是非法数据，如数字列的中间夹杂着一些汉字或者是符号；第二种是异常数据，异乎寻常的大数值或者是小数值。

```
def swap(x):
    if type(x) == str:
        if x[-1] == '岁':
            x = int(x[:-1])
        elif x[-1] == '分':
            x = int(x[:-1])
    return x
data = [['Alex', numpy.nan, '89 分'], ['Bob', '25 岁', '90 分'], ['Bob', '28 岁',
'90 分']]
df = pandas.DataFrame(data, columns=['Name', 'Age', 'Score'])
df = df.applymap(swap)
```

（5）数据格式清洗

```
data = [['Alex', 23, 80], [' Bob ', 25, 90], [' Bob', 25, 90]]
df = pandas.DataFrame(data, columns=['Name', 'Age', 'Score'])
# 大小写转换
df['Name'] = df['Name'].str.lower()
# 清除字符串前后空格
df['Name'] = df['Name'].map(str.strip)
```

（6）数据替换

```
data = [['Alex', 23, 80], ['Bob', 25, 90], ['Bob', 25, 90.5]]
df = pandas.DataFrame(data, columns=['Name', 'Age', 'Score'])
df['Name'] = df['Name'].replace("Bob", "Bauer")
```

替换时，字符串前后不能有空格存在，必须严格匹配。

6、数据处理

（1）排序

A、按标签排序

```
sort_index(self, axis=0, level=None, ascending=True, inplace=False,
            kind='quicksort', na_position='last', sort_remaining=True,
            by=None)
```

使用 sort_index()函数，通过传递 axis 参数和排序顺序，可以对 DataFrame 进行排序。 默认情况下，按照升序对行标签进行排序。通过将布尔值传递给升序参数 ascending，可以控制排序顺序；通过传递 axis 参数值为 1，可以对列标签进行排序。 默认情况下，axis = 0，对行标签进行排序。

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
', 'D', 'C'])
# 对列标签进行排序
df.sort_index(ascending=True, axis=1,inplace=True)
# 对行进行排序
df.sort_index(ascending=False, inplace=True)
```

B、按值排序

```
sort_values(self, by, axis=0, ascending=True, inplace=False,
             kind='quicksort', na_position='last')
```

使用 sort_values 函数可以按值排序，接收一个 by 参数，使用 DataFrame 列名称作为值，根据某列进行排序。

sort_values()提供 mergesort，heapsort 和 quicksort 三种排序算法，mergesort 是唯一的稳定排序算法，通过参数 kind 进行传递。

```
dates = pandas.date_range('20210101', periods=6)
df = pandas.DataFrame(numpy.random.randn(6, 4), index=dates, columns=['A', 'B
```

```
', 'D', 'C'])
# 按列AD进行排序
df.sort_values(ascending=True, by=['A', 'D'], inplace=True)
```

（2）分组

Pandas 可以使用 groupby 函数对 DataFrame 进行拆分，得到分组对象。

```
df.groupby(self, by=None, axis=0, level=None, as_index=True, sort=True,
           group_keys=True, squeeze=False, observed=False, **kwargs)
```

by:分组方式，可以是字典、函数、标签、标签列表

```
data = [['Alex', 24, 80], ['Bob', 25, 90], ['Bauer', 25, 90], ['Jack', 26, 80]]
df = pandas.DataFrame(data, columns=['Name', 'Age', 'Score'])
# 按Name分组
group = df.groupby(by="Name")
for k, v in group:
    print(k, v)
```

（3）合并

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left
_index=False, right_index=False, sort=True)
```

合并两个 DataFrame 对象。

left：左 DataFrame 对象

right：右 DataFrame 对象

on：列(名称)连接，必须在左 DataFrame 和右 DataFrame 对象中存在(找到)。

left_on：左侧 DataFrame 中的列用作键，可以是列名或长度等于 DataFrame 长度的数组。

right_on：来自右 DataFrame 的列作为键，可以是列名或长度等于 DataFrame 长度的数组。

left_index：如果为 True，则使用左侧 DataFrame 中的索引(行标签)作为其连接键。

right_index：与右 DataFrame 的 left_index 具有相同的用法。

how：可选值为 left，right，outer，inner，默认为 inner。left 是 LEFT OUTER JOIN，使用左侧对象的键；right 是 RIGHT OUTER JOIN，使用右侧对象的键；outer 是 FULL OUTER JOIN，使用键的联合；inner 是 INNER JOIN，使用键的交集。

sort：按照字典顺序通过连接键对结果 DataFrame 进行排序。默认为 True，设置为 False 时，可以大大提高性能。

```
data1 = [[1, 'Alex', 24, 80], [2, 'Bob', 25, 90], [3, 'Bauer', 25, 90]]
left = pandas.DataFrame(data1, columns=['ID', 'Name', 'Age', 'A'])
data2 = [[1, 'Alex', 87, 78], [4, 'Bob', 67, 87], [3, 'Bauer', 98, 78]]
right = pandas.DataFrame(data2, columns=['ID', 'Name', 'B', 'C'])
# 按 ID,Name 列进行合并
df = pandas.merge(left=left, right=right, on=['ID', 'Name'])
# 按 ID 列进行合并
df = pandas.merge(left=left, right=right, on='ID', how='left')
df = pandas.merge(left=left, right=right, on='ID', how='right')
df = pandas.merge(left=left, right=right, on='ID', how='outer')
```

（4）级联

```
concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
         keys=None, levels=None, names=None, verify_integrity=False,
         sort=None, copy=True)
```

沿某个轴进行级联操作。

objs 是 Series、DataFrame 或 Panel 对象的序列或字典。

axis 是操作的轴，默认为 0，axis=0 表示按 index 进行级联，axis=1 表示按 columns 进行级联。

join 是连接类型，默认 inner，指示如何处理其它轴上的索引。

sort 是是否进行排序，True 会进行排序，False 不进行排序。

Pandas 提供了连接 DataFrame 的 append 方法，沿 axis=0 连接。

```
pandas.append(self, other, ignore_index=False,
         verify_integrity=False, sort=None)
```

向 DataFrame 对象中添加新的行，如果添加的列名不在 DataFrame 对象中，将会被当作新的列进行添加。

other 是操作其它对象，可以是 DataFrame、series、dict、list

sort 是决定是否排序，boolean，默认是 None。

Pandas 提供了连接 DataFrame 的 join 方法，沿 axis=1 连接，用于将两个 DataFrame 中不同的列索引合并成为一个 DataFrame。

```
df.join(self, other, on=None, how='left', lsuffix='', rsuffix='',
        sort=False)
```

join 方法提供 SQL 的 Join 操作，默认为为左外连接 how=left。

```
data1 = [['Alex', 24, 80], ['Bob', 25, 90], ['Bauer', 25, 90]]
one = pandas.DataFrame(data1, columns=['Name', 'Age', 'A'])
data2 = [['Alex', 87, 78], ['Bob', 67, 87], ['Bauer', 98, 78]]
two = pandas.DataFrame(data2, columns=['Name', 'B', 'C'])
df = pandas.concat([one, two], axis=1, sort=True)
df = one.append(two, sort=True)
df = one.join(two, how="right", lsuffix='Left')
```

（5）迭代

DataFrame.iteritems()用于迭代(key，value)对，将每个列标签作为 key，value 为 Series 对象。

（6）SQL 化操作

```
data1 = [[1, 'Alex', 24, 80], [2, 'Bob', 25, 90], [3, 'Bauer', 25, 90]]
df = pandas.DataFrame(data1, columns=['ID', 'Name', 'Age', 'Score'])
# 查询指定列
print(df[['ID', 'Name']])
# 按条件查询
print(df[df['Name'] == 'Bauer'].head(5))
print(df[df['Score'] >= 90].head(5))
```