Python 学习笔记

Ysong

目录

1 Basic grammar

1.1 type

Listing 1.1: variable type

```
>>> str = "python"
>>> dir(str) #获取相关用法
>>> type(str) #获取变量类型
>>> help(str) #获取 guide
>>> id(str) #获取内存地址
>>> isinstance(str, string) #若 str 为 string 类型，则返回 True，否则返回 False
```

Listing 1.2: basic functions

```
print('Hello World!')
print('Hello', 'World!', sep=', ', end='') #各字符串间用 `, ' 分隔，结尾没有换行符
print(round(3.75, 1)) #四舍五入保留一位小数
float_number = 3.3; int_number = int(float_number)
```

1.2 operator precedence

Table 1.1: Python 运算符优先级表

优先级	运算符
1	() 函数调用
2	[] 下标
3	. 属性访问
4	** 指数运算
5	+, - 正负号
6	~ 按位非
7	*, /, //, % 乘, 除, 地板除, 取余
8	+, - 加, 减
9	<<, >> 左移, 右移
10	& 按位与
11	| 按位或
12	^ 按位异或
13	<, <=, >, >=, ==, != 比较运算符
14	= 赋值运算符
15	in not in 逻辑包含/不包含
16	not 逻辑非
17	and 逻辑与
18	or 逻辑或

“==”checks for equality, “is” checks for identity(sharing the same memory).

Listing 1.3: is & ==

```
"Jack" == "Jack" #return True
"Jack" is "Jack" #return False
```

无论是 x = x+1 还是 x += 1 都会重新给 x 分配内存。

1.3 condition statements

Listing 1.4: condition statement

```
def absolute_value(x):
  """Return the absolute value of x."""
  if x < 0:
    return -x
  elif x == 0:
    return 0
  else:
    return x

def absolute_value(x):
  return -x if x < 0 else x

```

- False values in Python: False, 0, ' ', None
- True values in Python: Anything else

1.4 loop

Listing 1.5: iteration

```
i, total = 0, 0
while i < 3:
    total += i
    i += 1
else: #若没有 break 则执行
    print(i)

for i in range(3):
    total += i
else: #若没有 break 则执行
    print(i)

```

1.5 function

函数用于处理需要多次重复运行的同一任务。python 的函数默认返回 None。

1.5.1 定义函数

Listing 1.6: definition of functions

```
  def my_func(string1,string2):
    print(string1)
    print(string2)

  def my_func1(string1='Hello',string2): #可以指定默认值
    ''' 功能:打招呼
        string1:默认hello
        string2:对象名字 '''
    print(string1)
    print(string2)
    return 1

  def my_info(*args, **kwargs): #这里 *args 会创建一个元组，kwargs 会创建一个字典
    print(args)
    print(kwargs)

  #可以使用 yield 函数返回，yield 允许函数在返回一个值的同时保存状态，并在下一次继续执行
```

```
#带 yield 的函数是一个 iterator
def count_up_to(n):
    counter = 1
    while counter <= n:
        yield counter
        counter += 1

#调用带有 yield 关键字的函数时，它会返回一个生成器对象
gene = count_up_to(5)
print(next(gene))
for number in gene:
    print(number)
#当生成器函数不再遇到 yield 语句时，生成会终止

```

1.5.2 调用函数

以下是几种正确的调用方法

Listing 1.7: use of function

```
my_func('Hello','world')
str1='Hello';str2='world'
my_func(str1,str2)
my_func(string1=str1,string2=str2) #写明形式参数名称为关键字参数
my_func(str1,string2=str2) #允许前面的形参不写名称，后面的写明
my_func(string2=str2,string1=str1) #写明形式参数名时可以交换顺序
info_value = ['Math', 'Art']; dic_value = {'name': 'John', 'age': 22}
my_info('Math', 'Art', name='John', age=22)
my_info(*info_value, **dic_value)

```

以下是几种错误的调用方法

Listing 1.8: mistakes in using functions

```
my_func(string1=str1,str2) #前面写明形参名称，后面必须写
my_func(string2=str2,str1)

```

1.5.3 可变参数

可变参数是 Python 特有的设计，可变参数也被称为不定长参数，即传入函数中的实际参数可以是 0 个、1 个或多个。

Listing 1.9: variable parameters

```
def greet(*names): #这里会将输入接受并放在一个元组中
  print("Hello",end=' ')
  for item in names:
    print(item,end=' ')

if __name__=='__main__':
  greet('Tom','Jerry')

  param=['Tom','Jerry'] #也可以调用列表
  greet(*param) #这里不能写形参名

```

另一种方法会将输入以“形参名: 变量名”的形式存为一个字典。

Listing 1.10: other variable parameters

```
def greet(**names): #输入转为字典
  print('Hello')
```

```
    for key,value in names.items():
      if value=='male':
        print('Mr',end=' ')
      else:
        print('Miss',end=' ')
      print(key)

if __name__=='__main__':
    greet(Tom='male',Jerry='female') #直接调用

    dict={'Tom':'male','Jerry':'female'} #也可以用字典调用
    greet(**dict) #这里也不能用关键字参数

```

1.5.4 静态类型函数

Python 的函数默认是动态类型的，可以接受所有类型的输入。当然也可以将函数定为静态类型的 (和 C++ 类似)。

Listing 1.11: static functions

```
def square(number:int|float)->int|float:
    return num**2

```

1.5.5 变量的作用域

- 在函数内的定义的变量是局部变量，外部不能调用。
- 函数外部定义的变量是全局变量，可以在任意位置调用该变量。
- 函数内定义的变量可以通过 global 关键字定为全局变量。
- environment 就是 frame 的顺序，当调用某变量时，会从里向外逐层寻找该变量。
- 在一个函数中，一个变量名只能始终为一个全局变量，或者始终是一个局部变量。
- 在内部函数中的变量名前加上 nonlocal，则内部函数的该变量与外部函数的变量是同一个。

Listing 1.12: local and global

```
message='Hello'
def my_func():
    global message
    cnt = 0
    def in_func():
      nonlocal cnt
    message='World'
    cnt++
print(message) #输出'World'

#下面这段代码会报错
x = 'global x'

def test():
    print(x)
    x = 'local x'
    print(x)

test()
#解决这个报错的方法有两种，即函数内声明全局或不使用全局
#可以在第一个 print(x) 前面加上 global x
#也可以删去第一个 print(x)

```

1.5.6 High-order function

High-order function 是一类返回函数的函数，用于处理同类型的任务。

Listing 1.13: High-order function

```
        def make_adder(n):
          """Return a function that takes K and return K + N.

          >>> add_three = make_adder(3)
          >>> add_three(4)
          """
          def adder(k):
            return k + n
          return adder

        add_three = make_adder(3)
        print(make_adder(100)(4))

```

High-order function 在音频领域经常被使用，用于制作波形。

Listing 1.14: mario

```
        from wave import open
        from struct import Struct
        from math import floor

        frame_rate = 11025

        def encode(x):
            i = int((1 << 14) * x)
            return Struct('h').pack(i)

        def play(sampler, name='song.wav', seconds=2):
            out = open(name, 'wb')
            out.setnchannels(1)
            out.setsampwidth(2)
            out.setframerate(frame_rate)
            t = 0
            while t < seconds * frame_rate:
                sample = sampler(t)
                out.writeframes(encode(sample))
                t += 1
            out.close()

        def tri(frequency, amplitude=0.3):
            period = frame_rate // frequency
            def sampler(t):
                saw_wave = t / period - floor(t / period + 0.5)
                tri_wave = 2 * abs(2 * saw_wave) - 1
                return amplitude * tri_wave
            return sampler

        c_freq, e_freq, g_freq = 261.63, 329.63, 392.00

        def both(f, g):
            return lambda t: f(t) + g(t)

        def note(f, start, end, fade=0.01):
            def sampler(t):
                seconds = t / frame_rate
                if seconds < start:
                    return 0
                elif seconds > end:
                    return 0
                elif seconds > end - fade:
                    return (end - seconds) / fade * f(t)
                elif seconds < start + fade:
                    return (seconds - start) / fade * f(t)
                else:
```

```
                    return f(t)
            return sampler

        c, e, g = tri(c_freq), tri(e_freq), tri(g_freq)
        g_low = tri(g_freq / 2)
        z = 0
        song = note(e, z, z + 1/8)
        z += 1/8
        song = both(song, note(e, z, z + 1/8))
        z += 1/4
        song = both(song, note(e, z, z + 1/8))
        z += 1/4
        song = both(song, note(c, z, z + 1/8))
        z += 1/8
        song = both(song, note(e, z, z + 1/8))
        z += 1/4
        song = both(song, note(g, z, z + 1/8))
        z += 1/2
        song = both(song, note(g_low, z, z + 1/8))
        z += 1/2

        play(song)

```

1.5.7 self-reference

self-reference function 会返回自身或者与自身相关的函数，这样可以递归的执行不定长度的任务。

Listing 1.15: self-reference

```
def print_all(x):
  print(x)
  return print_all
print_all(1)(2)(3) #这里会全部输出来

```

1.5.8 decorator

decorator 是一种用于修改函数或方法行为的工具，它接受另一个函数作为输入，并返回一个新的函数。decorator 可以叠加多层

Listing 1.16: decorator

```
    def decorator_function(original_function):
        def wrapper_function():
            print('wrapper executed before {}'.format(original_function.__name__))
            return original_function()
        return wrapper_function

    @decorator_function #将装饰器应用到 display 函数上，使 display 自动被装饰器的逻辑包裹
    def display():
        print('display function ran')

    #display = decorator_function(display) 装饰器的效果和这一句相同

    display() #上面的两句都会打印

    #若待装饰的 function 是有参数的，则需要给装饰器的返回值也添加上参数
    def decorator_function(original_function)
        def wrapper_function(*args, **kwargs)
            print('wrapper executed before {}'.format(original_function.__name))
            return original_function(*args, **kwargs)
        return wrapper_function

    #decorator 也可以带有别的参数
```

```
        def prefix_decorator(prefix):
          def decorator_function(original_function):
              def wrapper_function():
                  print(prefix, 'wrapper executed before {}'.format(original_function.__name__))
                  return original_function()
              return wrapper_function
          return decorator_function

        @prefix_decorator('TESTING:')
        def display():
            print('display function ran')

```

1.5.9 function currying

function currying 将一个接受多个参数的函数分解为一系列每个只接受一个参数的函数，即一个函数链。

Listing 1.17: function currying

```
        def curry2(f):
          def g(x):
            def h(y):
              return f(x, y)
            return h
          return g

        from operator import add
        m = curry2(add)
        add_three = m(3)
        print(add_three(2))

```

1.5.10 匿名函数

Python 的匿名函数也就是 lambda 表达式。匿名函数可以是另一个函数的参数，如 sort 函数的 key 变量。

- 只有 def 得到的函数会有一个本征名，在 shell 中输入变量名即可看到这一差异。
- def 函数可以有多步运行过程，而 lambda 表达式只能有一个表达式。

语法为 result=lambda [arg1[arg2,···,argn]]:expression

Listing 1.18: lambda function

```
        result = lambda r:math.pi*r*r
        area = (lambda r: math.pi * r * r)(3)
        print(result(5))

        def area(r):
          return r * r * math.pi
        result = area

```

1.6 input and output

1.6.1 打开关闭文件

Listing 1.19: open and close files

```
file = open(<filename>[,mode]) #打开文件
print(file.name); print(file.mode) #输出文件名和打开方式
file.close() #关闭文件
print(file.closed) #检查文件是否已经关闭，已关闭会返回 True

#用 with 语句打开文件可以确保文件被正确关闭，且可以避免文件打开错误引起的异常
with open(<filename>[,mode]) as file:

```

Table 1.2: Methods of Opening a File

符号	说明	注意
r	只读模式，指针放在开头	文件必须存在
rb	二进制只读模式	
r+	可以读取，也可以从文件的开头覆盖	
rb+	二进制读写模式	
w	只写模式式	若文件存在，则覆盖，否则创建新文件
wb	二进制只写模式	
w+	只读模式打开文件并清空	
wb+	二进制只读模式并清空文件	
a	追加模式	若文件存在，则指针放在末尾，否则创建文件
ab	二进制追加模式	
a+	追加且可读 (注意指针位置)	
ab+	二进制追加且可读	

1.6.2 读取写入文件

Listing 1.20: read and write files

```
#读取文件
with open('test1.txt', 'r', encoding='utf-8') as file:
    f_contents = file.read(); f_contents = file.read(100) #读取文件前 100Byte 的内容
    f_list = file.readlines()) #按行切分并组成列表
    f_list = file.readline() #只读一行，结尾带有换行符
    for line in file:
        print(line, end='')遍历每一行
    file.tell() #返回 cursor 在文件中的位置
    file.seek(7) #移动 cursor 到第 7Byte 的位置
#写入文件
with open('test2.txt', 'w') as f:
    f.write('Test')
    f.seek(0); f.write('Hello World!') #这里的内容会覆盖原有的内容

#读写非文本文件，可以用二进制读写，下面以创建一个图片副本为例
with open('puppy.jpg', 'rb') as rf:
    with open('puppy.jpg', 'wb') as wf:
        chunk_size = 4096
        rf_chunk = rf.read(chunk_size)
        while len(rf_chunk) > 0:
            wf.write(rf_chunk)
            rf_chunk = rf.read(chunk_size)

```

read() 和 write() 函数还有以下可选参数：

- offset: 移动字符个数
- whence: 指定从什么位置开始计算，0 表示开头，1 表示当前位置，2 表示文件末尾 (只能在二进制模式下使用)

1.6.3 上下文管理器

上下文管理器就是可以在 with 语句中被调用的特殊函数，可以自动管理资源的分配和释放。利用 contextmanager decorator 可以将普通的函数转换成上下文管理器。

Listing 1.21: contextmanager

```
import contextlib

@contextlib.contextmanager
def open_file(file, mode): #自定义打开文件的函数
    try:
        f = open(file, mode) #yield 之前的语句将在进入 with 语块时执行
        yield f
    finally:
        f.close() #yield 之后的语句将在退出 with 语块时执行

with open_file('test.txt', 'w):
    f.write('Hello, world!')

import os

@contextlib.contextmanager
def change_dir(destination): #自定义切换目录的函数
try:
    cwd = os.getcwd()
    os.chdir(destination)
    yield
finally:
    os.chdir(cwd)

```

1.6.4 处理.csv 文件

csv 是一种表格类型的文件，其每行各单元之间有分隔符，默认分隔符是‘,’。若单元格内部也有域分隔符相同的字符，该单元格会带引号。

Listing 1.22: read and write .csv files

```
import csv

with open('test.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file) #csv_reader 是一个迭代器
    next(csv_reader) #跳过第一行
    for line in csv_reader:
        print(line) #这里每一行是一个 list

  dic_reader = csv.DictReader(csv_file) #以字典形式读取
  #若行数为 n，则得到一个长度为 n-1 的迭代器，第 1 行是 key，其余 n-1 行为 value

with open('new_names.csv', 'w') as new_file:
    csv_writer = csv.writer(new_file, delimiter='\t') #这里设定了分隔符
    csv_writer.writerow(list(range(5)))

    dic_writer = csv.DictWriter(new_file, fieldnames=['number', 'name', 'score'])
    #以字典形式写入，fieldnames 为 colomn 名，即第一行
    csv_writer.writeheader() #写入第一行的 fieldnames
    csv_writer.writerow({'number': 1, 'name': 'Jack', 'score': 98})

```

1.6.5 处理.json 文件

json 库可以用于处理.json 文件 (JavaScript Object Notation)。json 文件中的 object 会被处理称 dict，array 会被处理生 list，其他类型的变量也会对应处理。

Listing 1.23: json module

```
import json

data = json.loads(json_string) #将 json 文件的数据类型转换为 python 中的数据类型
new_json_string = json.dumps(data, indent=2, sort_key=true) #将 python 语句转换称 json 语句

with open('test.json') as f:
    data = json.load(f) #读取 json 文件的内容
with open('new_test.json', 'w') as wf:
    json.dump(data, f) #写入 json 文件
#可选:indent 设置缩进为 2, sort_key 设置排序所有键

```

1.6.6 处理 zip 文件

Listing 1.24: zip file

```
import zipfile

my_zip = zipfile.ZipFile('files.zip', 'w') #创建 zip file
my_zip.write('test.txt') #将已有的文件写入 zip file
my_zip.write('thumbnail.png')
my_zip.close()

with zipfile.ZipFile('files.zip', 'w') as my_zip: #使用 file manager
    my_zip.write('test.txt')
    my_zip.write('thumbnail.png')
with zipfile.ZipFile('files.zip', 'w', compression=zipfile.ZIP_DEFLATED) as my_zip: #压缩文件

with zipfile.ZipFile('files.zip', 'r') as my_zip: #压缩文件
    print(my_zip.namelist()) #获取 zip file 中的文件表
    my_zip.extractall('files') #提取文件到 files 文件夹下
    my_zip.extract('thumbnail.png') #提取一个文件

import shutil

shutil.make_archive('another', 'zip', 'files') #打包 files 文件夹到 another.zip
shutil.unpack_archive('another.zip', 'another') #加压 another.zip 到 another 文件夹
#压缩文件可选 zip,tar,gztar,bztar,xztar

```

2 Collections

2.1 iterator

iterator 和 generator(带 yield 的函数) 有些类似，区别在于内存的使用，iterator 会生成所有的值并存在内存里，generator 则按需生成值，节省内存。例如，当需要读取一个大文件中的前几行时，iterator 可以避免吧整个文件都读取进来。

Listing 2.1: iterator

```
i_nums = iter([1, 2, 3, 4, 5]) #从 iterable 生成 iterator
ite = map(lambda x: x * x, range(10), [i for i in range(20)])
ite = map(pow, range(10), itertools.repeat(2))
ite = filter(lambda x: x % 2 == false, range(10))
#这两个函数里第二个及更后面的参数可以是 iterator,range,tuple,list,string 等可以迭代的内容
#map 会对遍历过程中的每个元素进行 function 操作，filter 只会留下 condition 为 True 的元素
ite = zip('abcd', range(4), [i*i for i in range(4)])
#zip 返回的迭代器是各可循环容器的元素组成的元组，遍历直到有容器被遍历完
ite = zip_longest('abcd', range(4), [i*i for i in range(4)])
#这里会一直遍历到所有 iterable 都遍历完，用 None 补充没有的部分
print(next(result)); print(next(result)) #逐项查找
for value in ite:
  print(value) #遍历迭代器
```

```
#注意，迭代器在遍历后会变为空，这一点与 list,range,tuple 不同。
#迭代器转为 list 等其他变量类型时会将其遍历一遍，因此同样会变空。

```

2.1.1 itertools

itertools 可以处理 iterator。

Listing 2.2: itertools

```
counter = itertools.count(5) #返回一个从 5 到无穷的 iterator
counter = itertools.cycle([1, 2, 3]) #返回列表循环的 iterator
counter = itertools.repeat(2, times=3) #返回 2 重复 3 次的 iterator, times 默认无穷
counter = itertools.starmap(pow, [(0, 2), (1, 2), (2, 2)]) #与 map 类似
counter = itertools.combinations("abcd", 2) #两两无序组合为 tuple, 组成 iterator
counter = itertools.permutations("abcd", 2) #两两有序组合为 tuple, 组成 iterator
counter = itertools.combinations_with_replacement("abcd", 4) #允许 tuple 中元素重复
counter = itertools.permutations_with_replacement("abcd", 4) #允许 tuple 中元素重复
combined = itertools.chain("abcd", [1, 2, 3]) #拼接 iterables, 类似 zip
result = itertools.islice(range(10), 1, 5, 2) #从 index=1 迭代到 index=4, step=2
result = itertools.compress("abcd", [True, True, False, True]) #abd 组成的 iterator
result = itertools.filterfalse(lambda x: x%2 == 0, range(10)) #判断与 filter 相反

students = [
  {
    "name": "Jack",
    "city": "Shanghai",
    "group": 1,
  },
  {
    "name": "Amy",
    "city": "Beijing",
    "group": 2,
  },
  {
    "name": "Jerry",
    "city": "Hangzhou",
    "group": 1,
  },
  {
    "name": "Jane",
    "city": "Wuhan",
    "group": 2,
  }
]
students.sort(key=lambda x: x["group"])
student_group = itertools.groupby(students, lambda x: x["group"])
for key, group in student_group: #这是一个二级 iterator
    print(key) #group 名, 即 list 中 group 的值
    for person in group:
      print(person)

```

2.2 list

Listing 2.3: basic operations of list

```
#创建列表
empty_list = []; empty_list = list()
student1=["Hermione","Harry","Ron"]
student2=["Draco","Padma"]
student_score=list(range(10)) #[range(10)] 会创建一个只有 range(10) 一个元素的列表
student_score=[(letter, num) for letter in 'abcd' for num in range(4)]
score = [(letter, num) for letter, num in zip('abcd', range(4))] #注意这两个是不同的
student_score = list(map(lambda n: n*n, range(10)))
#对列表元素进行操作
```

```
        student="Potter"
        student1.append(student) #在列表的末尾追加
        student1.insert(2, student) #在指定 index 处插入
        student2.extend(student1) #列表的拼接，这样可以比 + 更快
        del student[-1] #删除最后一个元素
        student_popped=student1.pop(1) #删除指定 index 处的元素，默认删除最后一个
        student_popped=student1.remove('Padma') #寻找第一个指定元素并删除，找不到会报错
        student_reversed=student1.reverse() #反转元素顺序
        seq_Ron=student.index("Ron") #查找第一个匹配的元素并输出，找不到会报错
        #列表拼接、扩展、排序、遍历
        student1=student1+student2
        student3=sorted(student1); student1.sort() #sorted 是一个 function，sort 是一个 method
        student3=sorted(student1, key=str.lower(), reverse=True) #不区分大小写倒序排序
        #sorted 函数会对每个元素进行 key 的操作，按照返回值的顺序，对原数据进行排序
        student1.sort(reverse=True) #True stands for down
        if "Hermione" in student1:
          print("Hermione is in student1")
        for a in student1: #遍历列表
        for index, item in enumerate(student1):
        for index, student in enumerate(student1, start=1): #这里 index 可以从 1 开始编号
        students = ', '.join(student1) #将 list 中的元素合并成一个字符串，以逗号为分隔符
        #列表统计
        number=student1.count("Padma") #统计元素出现的数量
        ind=student2.index("Padma") #找到元素首次出现的位置
        min_one, max_one = min(student_score), max(student_score)
        total=sum(student_score) #列表求和

```

2.3 tuple

元组是不可变的序列，但可以重新赋值

Listing 2.4: basic operation of tuple

```
        #元组创建
        empty_tuple = (); empty_tuple = tuple() #创建空元组
        a=tuple(rage(10,20,3)); a=(); a=(1,2,3)
        a_tup = sorted(a) #tuple 没有 sort method

```

2.4 set

Python 的 set 是无序集合，这与 C++ 不同。

Listing 2.5: basic operation of set

```
        students=[
          {"name":"Hermione","house":"Gryffindor"},
          {"name":"Harry","house":"Gryffindor"},
          {"name":"Ron","house":"Gryffindor"},
          {"name":"Draco","house":"Slytherin"},
          {"name":"Padma","house":"Ravenclaw"},
        ]
        houses=set() #创建空集合，注意 houses= 会创建一个字典
        houses.pop() #删除第一个元素，注意顺序是随机的
        houses.clear() #清空集合
        for student in students:
          houses.add(student["house"])
        for house in sorted(houses):
          print(house)
        if "Slytherin" in houses:
          print("Slytherin is in houses.")

```

2.5 dict

Listing 2.6: basic operation of dictionary

```
#创建字典
students1={
  "Harry":"Gryffindor",
  "Ron":"Gryffindor",
  "Draco":"Slytherin",
  "Padma":"Ravenclaw",
}
name=["Harry","Ron","Draco","Padma"]
house=["Gryffindor","Gryffindor","Slytherin","Ravenclaw"]
student1=dict(zip(name,sign))
student1=dict(n: s for n, s in zip(name,sign))
student1=dict((("Harry","Gryffindor"),("Ron","Gryffindor"),
    ("Draco","Slytherin"),("Padma","Ravenclaw"))) #这三句的效果相同
student1=dict(Harry="Gryffindor",Ron="Gryffindor",Draco="Slytherin",Padma="Ravenclaw")
student_empty=dict.fromkeys(name) #创建值为空的字典

#访问，排序，删除，加入，遍历
ron_house=student1["Ron"] #访问字典，找不到会报错
ron_house=student1.get("Ron") #访问字典，找不到会返回 None，可以通过这种方法避免报错
david_house=student1.get("David", "Not Found") #访问字典，找不到返回 Not Found
student1["Hermione"]="Gryffindor" #加入元素
student1.update({"Hermione":"Gryffindor"}) #合并两字典

s_house = sorted(student1) #根据 key 来排序
ron_house=student1.pop("Ron") #删除元素并返回值
del student1["Ron"] #删除元素

for key,value in student1.items() #遍历字典，注意 item 是函数，要加 ()
for key in student1.keys(); for key in student1 #遍历键数组
for value in student1.values() #遍历值数组

```

2.6 String

2.6.1 编码解码

Listing 2.7: create a string

```
  raw_string = r'\tTab' #这里的\t 不会被替换
  name="Harry"; text='Hello world!'
  byte=name.encode('GBK') #编码，返回编码的 16 进制值
  name1=byte.decode('GBK') #解码

```

2.6.2 常用操作

Listing 2.8: basic operations of string

```
  length=len(name) #返回字符串长度
  size=len(name.encode()) #返回内存大小
  text.split(' ') #切分字符串
  strnew=string.join(house) #合并字符串
  sub='ab'
  str_replace=sub.replace('a', 'c') #字符串替换
  num=string.count(sub) #检索子字符串出现次数
  start=string.find(sub) #子字符串首次出现的索引，若没有出现，返回-1
  str_lower=str.lower() #转为小写，不会改变原字符串
  str_upper=str.upper() #转为大写
  str_strip=str.strip(['@']) #删去字符串左右两侧的' ',\t,\r,\n 等 (默认)，以及'@'(可选)
  str_lstrip=str.lstrip(['@']) #删去字符串左侧的特定字符
  str_rstrip=str.rstrip(['@']) #删去字符串右侧的特定字符
```

```
        str_filled = str.ljust(10, fillchar=' ') #字符串左对齐，右侧填充至 10 个字符
        str_filled = str.rjust(10, fillchar=' ') #字符串右对齐
        str_filled = str.center(10, fillchar='*') #字符串中间对齐
```

2.6.3 格式化字符串

格式化字符串也就是先制定一个模板，并在模板中以占位符为变量留下位置。一种实现方式是与 C 语言相似的占位符，更适合 python 的做法是使用 format 函数或者 f 字符串。

Listing 2.9: formatting string1

```
        '%[-][+][0][m][.n]格式字符'%exp #这是模板
        template = '编号：%09d\t 公司名称：%s\t官网：http://www.%s.com'
        print(template%(7, '百度', 'baidu'))
```

可选项如下

- -: 左对齐，正数前面无负号，负数前面有负号
- +: 右对齐，正数前面加正号，负数前面加负号
- 0: 右对齐，正数前面无负号，负数前面加负号，用 0 补足位数
- m: 占有宽度
- .n: 小数位数

Table 2.1: placeholder

格式字符	说明	格式字符	说明
%s	字符串	%c	单个字符
%x	十六进制整数	%o	八进制整数
%f %F	浮点数	%d %i	十进制整数
%%	字符%	%e	指数 (基底为 e)

format 函数使用模板

Listing 2.10: format string

```
        '{[index][:[fill][align][sign][#][width][.precision][type]]}'.format(item1, item2)
```

- index: 可选，表示该位置填入的是 format 的那个参数 (0-base)
- fill: 可选，填充字符
- align: 可选，< 左对齐，> 右对齐，^ 内容居中
- sign: 可选，+-, 用法与占位符选项相同
- #: 可选，进制前缀显示，也可以选 ‘,’ 表示添千位分隔符
- width: 可选，字段宽度
- .precision: 可选，小数位数

- type: 可选，指定类型

Table 2.2: type options

格式字符	说明	格式字符	说明
S	字符串类型 (默认)	b	十进制整数转为二进制
D	十进制整数	o	十进制整数转为八进制
C	按 ASCII 码转为字符	x X	十进制整数转为十六进制
e E	转为科学计数法	f F	转为浮点数，默认 6 位小数
g G	自动切换	%	转为百分数

下面给出一些例子

Listing 2.11: examples of format string

```
number, name, url = 7, "百度", "baidu"
template='编号:{:0>9s}\t公司名称:{:s}\t官网:http://www.{:s}.com'
context=template.format(number, name, url)
context_f=f'编号:{number}\t公司名称:{name}\t官网:http://www.{url}.com'

f_template='编号:{f_number}\t公司名称:{f_name}\t官网:http://www.{f_url}.com'
f_context = f_template.format(f_number = number, f_name = name, f_url = url)

context='编号:{2}\t公司名称:{0}\t官网:http://www.{1}.com'.format(name, url, number)

#下面假设有 company 变量，他有三个成员，分别是 name、url 和 number
context='编号:{0.number}\t公司名称:{0.name}\t官网:http://www.{0.url}.com'.format(company)

company = {'name': 'baidu', 'number': 7, 'url': 'baidu'}
context = '编号:{number}\t公司名称:{name}\t官网:http://www.{url}.com'.format(**company)
context = 'This function ran with args: {}, and kwargs: {}'.format(args, kwargs)
#这里 {} 中的内容可以是表达式或其他类型的数据 (如 list, dict)

```

2.7 collection

collection 包含了 dict，list，str 等 Container，同时提供了一些额外的数据结构，相当于 C++ 标准库。

Listing 2.12: namedtuple

```
import collection

#namedtuple 可以通过给 tuple 中每个变量加名称增加代码的可读性。
Color = collection.namedtuple('Color', ['red', 'green', 'blue'])
color = Color(55, 155, 255); print(color[0], color.red)

#deque 双端队列，可以从两端添加和删除元素
collection.deque([1, 2, 3])

#Counter 计数器，计算可迭代对象中元素的频率
c = collection.Counter(['a', 'b', 'c', 'a', 'b', 'b'])
c.update(['a', 'b']) #继续计数
c.most_common(5) #输出最多的 5 项，tuple 组成的 list

#OrderedDict 有序字典，记住元素插入顺序
collection.OrderedDict([('a', 1), ('b', 2)])

#defaultdict 可以为字典中没有的元素添加默认值，默认值类型为 int
dd = collection.defaultdict(int); print(dd['key']) #输出 0

#heapq 是优先级队列，用最小堆实现
heap = collection.heapq.heapify([10, 17, 50, 7, 30, 24, 27, 45, 15, 5, 36, 21])
```

```
        collection.heappush(heap, 13) #插入元素，heap 会以满二叉堆形式存储
        print(heapq.heappop(heap)) #弹出堆顶元素并返回

        c = Counter('hello world!'); c = Counter(['cat', 'dog'])
        d = Counter({'red': 3, 'blue': 2}); d = Counter(cat=4, dogs=8)
        print(c['cat'], c.items())
        ite = c.elements() #返回一个迭代器
        print(c.most_common(3)) #出现最多的三个
        print(c.total()) #总数
        c.subtract(d) #两个 counter 相减

```

2.8 SQLite

sqlite3 是 python 标准库中的数据库模块。sqlite3 中数据类型有 5 种，分别是 NULL，INTEGER，REAL(float)，TEXT，BLOB。

Listing 2.13: SQLite

```
        import sqlite3

        #创建并连接数据库
        conn = sqlite3.connect(':memory:') #将数据存在 RAM 中，每次关闭都会清空
        conn = sqlite3.connect('my_database.db') #将数据存在文件中

        c = conn.cursor() #创建指针，数据库中的内容将由 cursor 完成

        c.execute("""CREATE TABLE IF NOT EXISTS students( #创建名为students的表，并创建3个column
                    first_name text, #不能创建已经存在的column否则会报错
                    last_name text,
                    id integer
                    )""") #IF NOT EXISTS 是可选的

        #向表中添加数据
        c.execute("INSERT INTO students VALUES ('Amy', 'Elisabeth', 11)")
        c.execute("INSERT INTO students VALUES ('{}', '{}', {})".format( #用 f-string 格式化
            student1.first_name, student1.last_name, student1.id))
        c.execute("INSERT INTO students VALUES (?, ?, ?)", #可以用一个 tuple 简化操作
            (student1.first_name, student1.last_name, student1.id))
        c.execute("""INSERT INTO students VALUES
                (:first_name, :last_name, :pay)""", #可以用 dict 使操作更清晰
            {'first_name': student1.first_name,
            'last_name': student1.last_name,
            'id': student1.id})
        #从表中读取数据，* 表示提取全部
        c.execute("SELECT * FROM students WHERE last_name='Elisabeth'")
        c.execute("SELECT * FROM students WHERE last_name=?", ('Elisabeth',)) #1 个元素也要加 `,'
        c.execute("SELECT * FROM students WHERE last_name=:last", {'last': 'Elisabeth'})
        get_data = c.fetchone() #从 select 得到的内容中读取 1 行
        get_data = c.fetchmany(5) #从 select 得到的内容中读取 5 行，返回一个列表
        get_data = c.fetchall() #从 select 得到的内容中读取所有行

        conn.commit() #需要提交 cursor 的操作
        conn.close() #关闭数据库

        #with 语句可以让连接，提交等操作更加简便，下面给出两个例子
        conn = sqlite3.connect('my_database.db') #将数据存在文件中
        with conn:
            c.execute("INSERT INTO students VALUES ('Amy', 'Elisabeth', 11)")
            #离开 with 代码块时自动执行 conn.commit()

        with sqlite3.connect('my_database.db') as conn:
            c = conn.cursor
            c.execute("INSERT INTO students VALUES ('Amy', 'Elisabeth', 11)")
            #离开 with 代码块时自动执行 conn.commit() 和 conn.close()

```

3 Class

3.1 定义类模板

3.1.1 基本结构

Listing 3.1: define a class

```
class User:
    '''这是一个学生用户类''' #类的说明
    #以下是静态成员变量，Python 中成为类的属性
    student_user = 'Student User'
    student_number = 0

    def __init__(self, name, number): #构造函数
        self.__name=name #这里名字设置为私有成员，这里变量是非静态成员，python 称之为实例属性
        self.number = number;
        User.student_number += 1
        #这里的 name，number 是非静态成员变量，Python 中成为实例的属性

    def __str__(self): #输出运算的重载
        return f'{self.number} {self.__name}'

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        self.__name = name

    def introduce(self): #self 类似 *this 指针
        print("I'm ", self.__name, ", my number is ", self.number, sep='')

    @staticmethod
    def greet(input_student): #静态成员函数
        print(f"Hello, my name's {self.__name}!")

    @classmethod
    def get_info(cls): #类方法，其第一个变量是类，常用于类型转换，类属性操作等
        print(f"There are {cls.student_number} {cls.student_user} in total.")

```

3.1.2 访问限制

Python 没有对属性和方法的访问权限进行限制。为了保证类内部某些属性不被外部访问，可以在属性或方法名前 (或前后) 加上双下划线。

- __foo__: 首尾双下划线表示定义特殊方法，一般是系统定义名字。
- __foo: 双下划线表示私有成员, 只允许所在的类调用。
- _foo: 单下划线表示保护成员，即 C++ 中的 protected 成员。

3.1.3 属性

Python 中，数据成员被称为属性。可以通过 @property 将一个方法转换为属性，转换后可以直接通过方法名调用该方法，无需添加 ()。这样做可以简化代码，也为属性添加安全保护, 即添加了 @property 的属性是只读的 (这是由于 return 时经过了复制传递)。

Listing 3.2: property of class

```
class User:
    ''' 用户类 ''' #类的说明
```

```
    @property
    def name(self);
        return self.__name #这个 name 属性被设定为只读的

```

@name.setter 可以在给属性 name 赋值时运行

Listing 3.3: setter of class

```
@name.setter #这里 name 是属性名称
def name(self, name)
    if len(name) < 20:
        self.__name=name

```

@name.deleter 属性可以定义一个在删除 name 属性时执行的函数。

Listing 3.4: deleter

```
@name.deleter
def name(self):
    print('Delete Name!')
    self.__name = None
del student1.__name

```

3.1.4 类的特殊成员 (魔术方法)

Listing 3.5: Special members for class

```
print(User.__class__) #类的名称
print(User.__bases__) #类的基类
print(User.__dict__) #类的数据成员及对应的值，输出一个字典

def __new__(cls, *args, **kwargs): #创建类的实例
    print(f"Run new with: cls={cls}, args={args}, kwargs={kwargs}")
    return super().__new__(cls) #调用父类的 __new__ 方法，并传入当前类作为参数

def __del__(self): #对象删除时调用的方法

def __init__(self, name, number): #接收并初始化实例
    self.__name=name #这里名字设置为私有成员，这里的变量是非静态成员，python 中称之为实例属性
    self.number = number;
    User.student_number += 1
    #这里的 name，number 是非静态成员变量，Python 中成为实例的属性
    #Python 中，一个类函数只能有一个 __init__ 函数，若需要多个构造函数，可以用 classmethod 实现
#在实例创建时，会先调用 __new__ 在调用 __init__

def __str__(self): #print() 和 str() 的重载
    return f'{self.number} {self.__name}'
def __repr__(self): #返回一个 Python 合法的字符串，使之可以用 eval 重建
    return(f"User({self.__name}, {self.number})")
    #The goal of __repr__ is to be unambiguous
    #The goal of __str__ is to be readable
student1 = User("Amy", 11); repr_str = repr(student1)
copy_student1 = eval(repr_str)
def __format__(self, format_spec): #重载 str.format() 和 f-string

def __bool__(self): #bool() 的重载

def __add__(self, other): #重载 +
def __iadd__(self, other): #重载 +=
def __sub__(self, other): #重载-
def __isub__(self, other): #重载-=
def __mul__(self, other): #重载 *
def __imul__(self, other): #重载 *=
def __truediv__(self, other): #重载/
def __itruediv__(self, other): #重载/=
```

```
        def __floordiv__(self, other): #重载//
        def __mod__(self, other): #重载%
        def __pow__(self, other): #重载
        def __eq__(self, other): #重载 ==
        def __ne__(self, other): #重载!=
        def __lt__(self, other): #重载 <
        def __le__(self, other): #重载 <=
        def __gt__(self, other): #重载 >
        def __ge__(self, other): #重载 >=

        def __len__(self): #重载 len()
        def __getitem__(self, key): #重载 [], 读取值
        def __setitem__(self, key, val): #重载 []=, 修改值
        def __delitem__(self, key): #重载 del [], 删除值
        def __iter__(self): #返回迭代器, 重载 iter()
        def __next__(self): #返回迭代器的下一个值, 重载 next()

        def __call__(self, x): #允许实例像 function 一样被调用
            return self.__name + x
        #既然 class 可以像 function 一样被调用, 那 class 也可以成为decorator
        class decorator_class:
            def __init__(self, original_function):
                self.original_function = original_function

            def __call__(self, *args, **kwargs):
                print('call method executed before {}'.format(self.original_function.__name__))
                return self.original_function(*args, **kwargs)
        @decorator_class
        def display():
            print('display function ran')

        def __enter__(self): #进入 with 语句时自动调用
        def __exit__(self, exc_type, exc_value, traceback):
        #离开 with 语句时自动调用 (正常和非正常退出都会调用)
        #三个参数分别是异常类型、异常值和异常回溯信息, 返回 True 会一直异常, 否则异常会继续传播

```

3.2 类的使用

3.2.1 创建类的实例

Listing 3.6: Create an instance of class

```
        student1 = User('Amy', 11)

```

3.2.2 类成员的使用

Listing 3.7: Using class members

```
        print(student.student_number); print(User.student_number) #都输出 1
        print(student.number); student1.number = 1 #这里 number 无法修改

        student1.introduce() #输出 I'm Amy, my number is 11
        User.introduce(student1) #效果与上一句相同

        User.greet(student1) #输出 Hello, my name's Amy!
        student1.greet(student1) #效果与上一句相同

        User.get_info() #输出 There are 1 Student User in total.
        student1.get_info() #效果与上一句相同
        print(student1) #这里会调用 __str__ 函数, 输出 11 Amy

```

3.3 继承

Python 的继承默认是公有继承。继承的内容有类的属性，实例方法，类方法，静态方法，特殊方法，私有成员不会被继承。类方法在继承时，会以子类作为第一个参数。

Python 中的所有类都继承自 builtins.object 类。

Listing 3.8: Class inheritance

```
class Student(User)
    def __init__(self, grade, name, number):
        super().__init__(name, number) #调用基类的构造函数
        self.__grade = grade

```

子类可以重写父类中的方法，这与 C++，Java 没什么区别，只是 Python 没有虚函数机制。

3.4 有关类的内置函数

由于 python 中的类在访问和添加成员的操作上限制较少，可以调用一些内置函数来检查这些类的合理性。

Listing 3.9: built-in functions about class

```
isinstance(object, classinfo) #检查 object 的类别是不是 classinfo
isinstance(student1, User) #返回 True

issubclass(cls, classinfo) #检查 cls 类是不是 classinfo 的子类
issubclass(Student, User) #返回 True

hasattr(object, name) #检查 object 是否有 name 属性
hasattr(student1, "number") #返回 True

callable(object) #检查 object 是否可以像函数一样被调用
callable(student1.name) #返回 False
callable(student1.greet) #返回 True

```

3.5 类的典型例子

3.5.1 文件管理类

文件管理类可以实现打开创建文件，读写文件，文件异常处理等工作

Listing 3.10: File management class

```
class Open_File():

    def __init__(self, filepath, mode):
        self.filepath = filepath
        self.mode = mode

    def __enter__(self):
        print(f"Entering the file: {self.filepath}")
        self.file = open(self.filepath, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, traceback)
        print(f"Exiting the file: {self.filepath}")
        if exc_type:
            print(f"An exception occurred: {exc_type}, {exc_value}")
        self.file.close()

with Open_File('test.txt', 'w') as f:
    f.write('Testing')

```

4 Basic module

4.1 install and import module

4.1.1 install and manage modules

可以通过 pip 安装和管理所需的 module。

Listing 4.1: download modules

```
pip install pandas  #安装 module
pip uninstall pandas  #卸载安装的 module
pip list #列出所有已安装的 module，可选-o 查看是否是最新版本
pip install -U pandas #更新 module
pip freeze --local | grep -v '^\-e' | cut -d = -f 1 | xargs -n1 pip install -U
pip search pandas #搜索指定的 module
pip freeze > requirements.txt #将项目所需的 module 整理成文档
pip install -r requirements.txt #将文件中的 module 全部安装下来

```

4.1.2 virtualenv

virtualenv 是指项目运行的 module 环境，这个环境可以不包含 global 环境中的所有 module。

Listing 4.2: setup virtualenv

```
pip install virtualenv
mkdir Environments
cd Environments
virtualenv project1_env -p /usr/bin/python3.10 #创建 virtualenv 环境，python 版本可选
source project1_env/bin/activate #activate virtualenv
pip install numpy #这里下载所需的 module 即可
pip install -r requirements.txt #由文件下载所有 module
pip freeze --local > requirements.txt #将 virtualenv 中的 module 整理成文档
deactivate #退出 virtualenv
rm -rf project1_env #删除已创建的 virtualenv 环境

```

4.1.3 pipenv

管理环境是很有必要的，若所有项目都在 base 环境中运行，package 的更新可能导致项目不可用。

pipenv 结合了 pip 和 virtualenv，可以用来管理代码运行环境。pipenv 由 packages(生产环境运行必须有的包)，dev-packages(开发环境所需的包)

Pipfile 列出了基本信息，Pipfile.lock 是决定性的，列出了具体信息。

Listing 4.3: pipenv

```
pipenv install requests #向 package 中添加 module，环境保存在 Pipfile 中
pipenv install pytest --dev #向 dev 中添加 module
pipenv uninstall requests #删除 module

pipenv install #安装 Pipfile 中的所有 module
pipenv install --ignore-pipfile #安装 Pipfile.lock 中的所有 module
pipenv install -r requirement.txt #利用文件加载环境
pipenv graph #列出环境内容，包括依赖关系
pipenv lock -r > requirements.txt #整理环境内容
pipenv lock #更新 pip.lock

pipenv --python 3.6 #修改 python 版本，直接修改 Pipfile 不会修改环境
pipenv --rm #删除环境，不会删除 Pipfile
pipenv install #由 Pipfile 重建环境
pipenv --venv #查看虚拟环境目录
pipenv check #检查是否能更新，Pipfile 和环境是否对应
```

```

pipenv shell #激活环境
pipenv run python #在环境里运行指令 (打开 python)
exit #退出环境

touch .env #这个文件可以设置独属于这个环境的环境变量

```

Listing 4.4: .env

```
SECRET_KEY="MySuperSecretKey" #局部环境变量

```

4.1.4 anaconda

anaconda 也可以用来管理 module 和环境，其效果相当于 pip+virtualenv。anaconda 的优势在于可以安装不属于 python 的 module，且可提供图形化管理功能。

注意，anaconda 和 pipenv 有冲突，不能一起使用。

Listing 4.5: anaconda

```
conda create --name my_app python=2.7 flask sqlalchemy
#创建一个名为 my_app 的项目，项目环境中带有 flask 和 sqlalchemy，python 版本为 2.7
conda activate my_app #激活环境，默认环境为 base
conda deactivate #退出环境
conda env list #列出所有已创建的环境
conda remove --name my_app --all #删除已有的环境
conda env export > environment.yaml #导出 environment 的内容
conda env export create -f environment.yaml #通过文件创建环境

```

有时我们需要记录环境与对应项目的目录，这是我们可以在环境目录 (可以通过 conda env list 查看) etc/conda/activate.d/env_vars.sh，etc/conda/deactivate.d/env_vars.sh，这两个文件分别会在 activate 和 deactivate 时自动运行。

Listing 4.6: activate.d/env_vars.sh

```
#!/bin/sh
export DATABASE_URI="postgresql://user:pass@db_server:5432/test_db"
#just add anything you need

```

Listing 4.7: deactivate.d/env_vars.sh

```
#!/bin/sh
unset DATABASE_URI

```

可以通过 ~/.bashrc 中加入以下函数以自动启用文件夹中的 environment.yaml。

Listing 4.8: conda_auto_env

```
function conda_auto_env() {
  if [ -e "environment.yaml" ]; then
    ENV_NAME=$(head -n 1 environment.yaml | cut -f2 -d " ")
    #Check if you are already in the environment
    if [[ $CONDA_PREFIX != *$ENV_NAME* ]]; then
        #Try to activate environment
        conda activate $ENV_NAME &>/dev/null
    fi
  fi
}

#export PROMPT_COMMAND="conda_auto_env;$PROMPT_COMMAND"
#$PROMPT_COMMAND 每次按下 enter 时都会运行
```

```
#若不需要，可以注释掉最后一行，并使用 conda_auto_env 手动调用环境

```

4.1.5 import modules

Listing 4.9: import module

```
import math
import math as ma
from math import sqrt
from math import *

```

4.1.6 view the functions in modules

Listing 4.10: view the functions in builtin module

```
import builtins
print(dir(builtins)) #这个查看方法对所有 module 都有效

```

4.1.7 circular import

circular import 是一个常见的错误。当 main.py import B.py 的其中部分时，B.py 整个文件都会被运行，若 B.py 同样 import main，那么 A.py 也会从头开始运行整个文件，而这时 main.py 会找不到其 import B.py 中的部分，这时就触发了 circular import。

4.2 builtins module

最基础的 module，包含了 print len range abs 等常用函数，TypeError ValueError KeyError IndexError 等常见异常，True False None 等内置常量，int str list dict set 等内置类型，不需要导入即可使用。

4.3 sys module

有关系统操作的 module

Listing 4.11: sys module

```
print(sys.path)
#系统路径列表，可以通过 append 追加 module 所在地址
#也可以追加在 /.bashrc 的 PYTHONPATH 里面 (作为环境变量)
#如果需要 import 的 module 是一个文件夹，需要保证文件夹中有一个'__init__.py' 文件
#'__init__.py' 文件会在 import 时运行，该文件可以为空，也可以导入文件夹中的子模块
print(sys.executable) #输出 python 的文件位置
print(sys.version) #输出 python 版本，用来检验编译器

```

4.4 os module

os module 可以实现操作系统相关的功能。

Listing 4.12: os module

```
#文件管理
os.getcwd() #输出工作区目录
os.chdir('/mnt/d/Desktop') #移动到指定目录
```

```
os.listdir() #输出当前目录下的所有文件及文件夹
os.mkdir('os_test') #在当前目录下新建，注意只能新建一层
os.makedirs('os_test/my_profile') #新建目录，允许多层
os.rmdir('os_test'); os.removedirs('os_test/myprofile') #删除目录
os.rename('test.txt', 'demo.txt') #重命名文件
print(os.stat('demo.txt')) #查看文件属性，如文件大小 (st_size)，最后一次修改时间 (st_mtime)
for dirpath, dirnames, filenames in os.walk(/mnt/d/Desktop/python) #递归的遍历目录下所有文件
    if f.endswith('.py'): #筛出所有 python 脚本文件
#dirpath 表示当前所在的目录，dirname 表示当前目录下的文件夹，filenames 表示当前目录下的文件名

os.environ.get('HOME') #返回 home 的地址，这里还可以查询其他环境变量，如 PATH
os.path.join(os.environ.get('HOME"), 'test.txt') #拼接目录，直接字符串相加容易遗漏或多加slash
os.path.exists('/mnt/d/test.txt') #返回一个bool值，即目录是否存在
os.path.basename('/mnt/d/test.txt') #返回`test.txt'这里无论路径是否存在都不会报错
os.path.dirname('/mnt/d/test.txt') #返回 `/mnt/d'
os.path.split('/mnt/d/test.txt') #返回 (`/mnt/d', 'test.txt')
os.path.splitext('/mnt/d/test.txt') #返回 (`/mnt/d/test',`.txt')

```

4.5 subprocess module

subprocess 可以用于在终端中执行任务。

Listing 4.13: subprocess module

```
subprocess.run('ls') #在 terminal 中运行 ls，若运行失败不会 abort
subprocess.run('ls', check=True) #在 terminal 中运行 ls，若运行失败会 abort
subprocess.run('ls', stderr=subprocess.DEVNULL) #在 terminal 中运行 ls，忽略所有 error

p1 = subprocess.run(['ls', '-la']) #捕获运行的结果
p1.args; p1.returncode; p1.stderr #指令，返回值 (0 表示没有出错，1 表示出错)，标准错误流

p1 = subprocess.run(['ls', '-la'], capture_output=True) #捕获运行的输出 (字节流)
print(p1.stdout.decode()) #返回标准输出
p1 = subprocess.run(['ls', '-la'], capture_output=True, text=True) #捕获运行的输出 (字符串)
print(p1.stdout); #返回标准输出

with open('output.txt', 'w') as f:
    p1 = subprocess.run(['ls', '-la'], stdout=f, text=True) #输出到文件中

p1 = subprocess.run(['cat', 'test.txt'], capture_output=True, text=True)
p2 = subprocess.run(['grep', '-n', 'test'], capture_output-True, text-True, input=[p1.stdout])
#传递输出和输入值，逐步完成 process

```

4.6 time module

time module 可以用于计时

Listing 4.14: time module

```
time.sleep(1) #将进程挂起 1s
t = time.time() #返回 linux 时间戳，单位都是秒
t1 = time.perf_counter(); t2 = time.perf_counter() #高精度时间计数器
t3 = time.process_time(); t4 = time.process_time() #高精度的 CPU 时间
print(t2 - t1) #计算时间差，会包含 sleep() 的时间
print(t4 - t3) #计算时间差，不会包含 sleep() 的时间

```

Listing 4.15: Timer Class

```
class Timer:  #@save
    """记录多次运行时间"""
    def __init__(self):
        self.times = []
        self.start()
```

```
    def start(self):
        """启动计时器"""
        self.tik = time.time()

    def stop(self):
        """停止计时器并将时间记录在列表中"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """返回平均时间"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """返回时间总和"""
        return sum(self.times)

    def cumsum(self):
        """返回累计时间"""
        return np.array(self.times).cumsum().tolist()

```

4.7 threading module

4.7.1 threading

threading module 用于处理多线程。

Listing 4.16: threading module

```
my_function(sec):
    time.sleep(sec)
    return "accomplished"
t1 = threading.Thread(target=my_function, args=(1,)) #创建一个线程
t2 = threading.Thread(target=my_function, args=(2,)) #创建一个线程
t1.start(); t2.start(); #启动线程，同时运行余下部分 (3 个线程)
t1.join(); t2.join(); #待 t1,t2 线程结束后再继续运行余下部分 (2 个线程)

```

4.7.2 concurrent.futures

Listing 4.17: concurrent.futures module

```
import concurrent.futures

with concurrent.futures.ThreadPoolExecutor() as executor:
    f1 = executor.submit(my_function, 1)
    f2 = executor.submit(my_function, 2) #这两个线程是并行的
    print(f1.result()); print(f2.result())

    results = [executor.submit(my_function, i) for i in range(10)]
    for f in concurrent.futures.as_completed(results):
        print(f.result()) #根据完成先后打印

    results = executor.map(my_function, [i for i in range(5)])
    for result in results:
        print(result) #根据提交先后打印

```

4.7.3 multiprocessing

Listing 4.18: multiprocessing module

```
        processes = []
        for _ in range(i):
            p = multiprocessing.Process(target=my_function, args=(i,))
            p.start()
            processes.append(p)
        for process in processes:
            process.join()

```

4.8 logging module

logging module 是 python 标准库中用于记录日志的模块，它可以在程序运行时输出各种级别的日志。

Listing 4.19: logging module

```
    #basicConfig 用于设置日志的基本信息，这会设置 root logger，影响所有的 logger
    logging.basicConfig(filename='app.log', filemode='w',
                level=logging.DEBUG, #默认是 WARNING
                format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                datefmt='%Y-%m-%d %H:%M:%S'
                    #format 的更多选项见LogRecord attributes
                )
    #注意，多次设置 root logger 只有第一个生效，这在导入文件时要特别注意

    #可以使用 handlers 将日志同时输出到多个文件或输出到控制台
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler("logfile.log"),   # 文件处理器
            logging.StreamHandler()               # 控制台处理器
        ]
    )

    #为了让每个文件有单独的 logging 文件和等级，可以在每个文件里单独设置 logger
    logger = logging.getLogger(__name__) #一般以 __name__ 作为 logger 的命名
    logger.setLevel(logging.INFO) #设置 logger 的 level
    file_handler = logging.FileHandler('test.log') #设置日志文件和格式
    file_handler.setFormatter(logging.Formatter('%(levelname)s:%(name)s:%(message)s'))
    logger.addHandler(file_handler) #将 logger 的内容输出到文件中
    string_handler = logging.StreamHandler()
    string_handler.setFormatter(logging.Formatter('%(levelname)s:%(name)s:%(message)s'))
    logger.addHandler(string_handler) #将 logger 的内容输出到终端

    #以下信息将会以上面设定的形式存到 app.log 文件中，只有 level 以上级别会显示出来
    logging.debug("这是调试信息")
    logging.info("这是一般信息")
    logging.warning("这是警告信息")
    logging.error("这是错误信息")
    logging.critical("这是严重错误信息")

    #若为每个文件单独设置了 logger，则可以这样调用
    logger.info("这是一般信息")

    #error 等级的错误可以添加报错信息，即 traceback
    logging.exception("这是一条带有Traceback的错误信息")

```

4.9 re module

4.9.1 regular expression

正则表达式由元字符，限定符，选择字符，排除字符等组成。

元字符中的 '.''$''^', 限定符中的 '?''+''*' 以及 '\' 若需要匹配，则应该使用转义字符。

Table 4.1: meta-character

元字符	匹配对象
.	换行符以外的任意字符 [^\n\r]
\w	字母、数字、下划线或汉字
\W	非字母、数字、下划线、汉字
\s	任意空白字符 ([\f\n\r\t\v])
\S	任意非空白字符
\d	数字字符
\D	非数字字符
\b	单词的边界 (开始或结束)
\B	非单词的边界
^	字符串的开始
$	匹配字符串的结束

Table 4.2: qualifier

限定符	含义
?	匹配 0-1 次
+	匹配至少 1 次, 会贪婪匹配
*	匹配 0 次或多次, 会贪婪匹配
+? *?	匹配规则同上，但是非贪婪
{n}	匹配 n 次
{n,}	匹配至少 n 次
{n,m}	匹配至少 n 次，至多 m 次
|	匹配左右其一

- 方括号表示字符集合，如 [aeiou] 匹配元音字符，[a-z] 匹配小写字符。[\u4e00-\u9fa5] 匹配汉字。
- 方括号内的 ^ 表示排除字符，如 [^ aeiou] 匹配非元音字母的所有字符。
- 方括号中的无法使用限定符，限定符都判定为原来的字符。
- 小括号可以改变限定符的作用范围，如 (thir|four)th 相当于 thirth|fourth，(\.[0-9]{1,3}){3} 会将括号中的内容重复三次。
- 小括号还可以基于匹配模式从字符串中提取子字符串，组成一个元组。

Python 中使用正则表达式需要将部分\转义, 由于需要转义的\可能很多，可以使用模式字符串，即在字符串前加上 r 或 R，例如:r'\bm\w*\b'。

4.9.2 match string

Python 中的 re 模块可以通过正则表达式处理字符串。
下面是常用的匹配函数，这里匹配的字符串区间没有重叠的部分。

Listing 4.20: string matching

```
1 import re #导入模块
2
```

```
        pattern=r'mr_\w+' #模式字符串
        string='MR_SHOP mr_shop' #待匹配字符串

        #match 方法可以从字符串开始处开始匹配，若匹配失败，返回 None
        match=re.match(pattern,string,re.I) #不区分大小写匹配字符串，返回一个 match 对象
        print(match.start());print(match.end()) #输出匹配字符串起始位置
        print(match.span()) #输出匹配字符串起止位置元组
        print(match.string) #输出匹配前的字符串，即 string
        print(match.group()) #输出匹配的数据

        #search 方法可以搜索字符串中第一个可以匹配的子串，返回一个 match 对象
        match=re.search(pattern,string,re.I) #不区分大小写匹配字符串，返回一个 match 对象

        #findall 和 finditer 方法可以搜索字符串中所有可以匹配的子串
        match = re.findall(pattern,string,re.I) #不区分大小写匹配字符串，返回一个 match 对象
        print(match) #输出匹配子串的列表
        match = re.finditer(pattern, string, re.I) #返回一个由 match 对象组成的迭代器

        #上面的代码可以通过编译模式字符串进行简化
        pattern = re.compile(r'mr_\w+', re.I)
        match = pattern.match(string)
        match = pattern.match(string)
        match = pattern.findall(string)
        match = pattern.finditer(string)

        #捕获组的使用
        pattern = re.compile(r'(\w+)-(\d+)-(\w+)')
        text = "abc-123-def"
        match = pattern.match(text) #这里使用方法与上面相同
        #match.group(0) 为'abc-123-def', match.group(i) 为第 i 个捕获组捕获的内容
        match = pattern.findall(text) #match = [(`abc', `123', `def')]

```

Table 4.3: matching pattern

标志	含义
A ASCII	只匹配 ASCII 范围内的字符
I IGNORECASE	不区分大小写
M MULTILINE	将 ^ 和 $ 用于每一行的开头和结尾
S DOTALL	. 匹配所有字符，包括换行符
X VERBOSE	忽略未转义的空格和注释

4.9.3 substitute string

sub 方法可以实现 vim 中批量搜索并替换字符串的操作。
模板为:re.sub(pattern,repl,string,count,flags)

- pattern: 模式字符串
- repl: 替换后的子字符串
- string: 原始字符串
- count: 可选，最大替换次数，默认为 0，表示替换所有的匹配
- flags: 可选，见Table 4.3

Listing 4.21: string substitution

```
        import re

        pattern = r'1[34578]\d{9}' #模式字符串
```

```
string='电话号码是:13611111111'
result=re.sub(pattern,'1xxxxxxxxx',string)

pattern = re.compile(r'(1[34578]\d)\d{8}')
string = 'The phone numbers are:13611111111,15888888888,18333333333'
result = pattern.sub(r'\1********', string) #这里\1 表示第一个捕获组
result = 'The phone numbers are:136********,158********,183********'
```

4.9.4 split string

re.split 方法可以根据正则表达式切分字符串，匹配正则表达式的子串将被当作分隔符，并将分割结果以列表的形式返回。

模板为:re.split(pattern,string,[maxsplit],[flags])

- pattern: 模式字符串
- string: 待切分的字符串
- maxsplit: 可选，最大拆分次数
- flags: 可选，见Table 4.3

Listing 4.22: string segmentation

```
import re

pattern=r'[?&]'
url='http://www.mingrisoft.com/login.jsp?username="mr"&pwd="mrsoft"'
result=re.split(pattern,url)
print(result)
```

4.10 date module

4.10.1 datetime module

datetime 用于时间和日期戳记录和转换

Listing 4.23: datetime & pytz

```
d = datetime.date(2020, 7, 24); print(d) #这里的 day 是一个 date 类型的变量
td = datetime.date.today() #输出今天的日期
print(d.year); print(d.month); print(d.day)
print(d.weekday()); print(tday.isoweekday())
#in weekday: Monday 0 Sunday 6, in isoweekday: Monday 1 Sunday 7
tdelta = datetime.timedelta(days = 7) #这里 tdelta 是 timedelta 类型的，可以参与加减法
date2 = date1 + timedelta; timedelta = date1 - date2
tdelta.total_seconds() #时间差转秒

t = datetime.time(9, 30, 45, 100000) #四个参数分别是 h m s μs
print(t.hour); print(t.minute); print(t.second); print(t.microsecond)
print(t.iosformat()) #以国际标准形式输出时间
print(t.strftime('%B %d, %Y')) #以自定义形式输出时间，这里可以参考strftime 官方文档

dt = datetime.datetime.strptime('July 26, 2016', '%B %d, %Y') #根据模板由 string 转 datetime
dt = datetime.datetime(2020, 7, 8, 12, 10, 32, 100000, tz = pytz.UTC) #时区可选
t = dt.time(); d = dt.date(); print(dt.year)
sentence = f'Jack has a birthday on {birthday:%B %d, %Y}' #结合 fstring 使用
```

4.10.2 pytz module

pytz 用于管理时区 (time zone) 相关的信息。

Listing 4.24: pytz module

```
        dt_today = datetime.datetime.today() #返回当前时区的时间
        dt_now = datetime.datetime.now(tz = pytz.UTC) #返回指定时区的时间，默认没有时区信息
        dt_utcnow = datetime.datetime.utcnow() #返回 utc 时间，不建议使用
        dt_mtn = dt_now.astimezone(pytz.timezone('Asia/Shanghai'))
        #需要带有时区参数的 datetime 变量才可以进行时区转换
        for tz in pytz.all_timezones:
          print(tz)  #输出所有的时区名称
        dt_mtn = pytz.timezone('America/New_York').localize(dt_today) #为时间添加时区
        dt_time = datetime.datetime.fromtimestamp(mod_time) #将时间戳转换成 datetime 类型

```

4.10.3 calendar module

calendar 用于计算天数，星期数，星期几等数据。

Listing 4.25: calendar module

```
        today = datetime.date.today()
        days_in_current_month = calendar.monthrange(today.year, today.month)
        #输出一个 tuple，第一个元素表示今天星期几 Monday0 Sunday6，第二个表示本月有几天

```

4.11 PIL module

PIL module 可以处理图片，module 中的具体内容见PIL 官方文档。

Listing 4.26: PIL

```
      from PIL import Image, ImageFilter

      image1 = Image.open('pup1.jpg')
      image1.save('pup1.png')
      image1.show() #显示图片

      image1.thumbnail((300, 300)) #压缩至 300*300 像素
      image1.rotate(90) #逆时针旋转 90°
      image1.convert(mode='L') #图片变黑白

      image1.filter(ImageFilter.GaussianBlur(15)) #模糊图片，模糊程度可选

```

4.12 random module

Listing 4.27: random module

```
      value = random.random() #生成 [0.0, 1.0) 的随机浮点数
      value = random.uniform(1, 10) #生成 [1.0, 10.0] 的随机浮点数
      value = random.randint(1, 10) #生成 [1, 10] 的随机整数
      value = random.choice(['Red', 'Greed', 'Blue']) #从 list 中随机取出一个
      results = random.choices(['Red', 'Greed', 'Blue'], k=10) #随机取 k 遍，输出一个 list
      results = random.choices(['Red', 'Greed', 'Blue'], weight=[18, 18, 2], k=10)
      shuffled_list = list(range(2, 30)); random.shuffle(shuffled_list) #随机打乱 list
      hand = random.sample(list(range(50)), k=5) #取 5 个样本，5 个样本不会重复
      random.shuffle(my_list) #直接打乱列表，没有返回值

```

4.13 secrets module

secrets 可以生成随机长密码。

Listing 4.28: secrets module

```
import secrets
print(secrets.token_hex(16)) #16Bytes

```

4.14 cn2an module

cn2an module 可以实现中文数字到阿拉伯数字的转换。

Listing 4.29: cn2an

```
def chinese_to_int(chinese_str):
    arabic_number = cn2an.cn2an(chinese_str, 'normal')
    return arabic_number

```

4.15 Progress Bar

4.15.1 tqdm module

Listing 4.30: tqdm module

```
from tqdm import tqdm
import time

for i in tqdm(range(500), desc="Processing"):
    time.sleep(0.01)

progress_bar = tqdm(total=2000, desc="Processing")
for i in range(200):
    for j in range(10):
        progress_bar.update(1)

```

5 Maths & Statistics

5.1 math

math module 提供了最基础的数学函数。

Listing 5.1: math module

```
math.ceil(val) #向上取整
math.floor(val) #向下取整
math.trunc(val) #向零取整
math.fabs(val) #取绝对值，只用于浮点数，abs 通用于整数、浮点数、复数等

math.sqrt(val) #开平方根
math.pow(x, y) #计算 `x' 的 `y' 次方
math.log(x, base); math.log10(x); math.log2(x) #计算对数，默认 base 为 e
math.e; math.exp(x) #得到 e 和 e 的乘方
math.sin(x); math.asin(x) #三角函数和反三角函数
math.sinh(x); math.asinh(x) #双曲函数和反双曲函数
math.gamma(x) #gamma 函数 Γ(x)

math.factorial(x) #返回 x 的阶乘
math.gcd(x, y) #返回 `x'`y' 的最大公约数
math.degrees(x); math.radians(x) #弧度和角度转换

```

5.2 numpy

5.2.1 定义矩阵

Listing 5.2: Define matrices using numpy

```
import numpy as np

a = np.array([0.1*i for i in range(100)])
a = np.array([[i+5*j for i in range(5)] for j in range(5)])
a = np.arange(25).reshape(5,5) #reshape 可以重新规定矩阵型号
a = np.linspace(0,10,100) #第三个参数是生成的列表长度
a = np.arange(0,10,0.1) #第三个参数是步长
a = np.logspace(0.9.10) #生成 10 的 0-9 次幂
a = np.eye(3) #生成三维单位阵
a = np.diag([1,2,3,4,5]) #生成对角阵
a = np.random.rand(2,3) #生成 2*3 随机矩阵,0-1 均匀分布
a = np.random.randn(2, 3)
a = np.random.random((2,3)) #生成 2*3 随机矩阵，用元组表示大小
a = np.random.randint(low,high,size=(2,3)) #生成 2*3 随机整数矩阵

```

5.2.2 特殊函数

Listing 5.3: Advanced operations in numpy

```
X,Y=np.meshgrid(x,y) #将 x,y 扩展为一个矩阵
x,y=np.outer(x,y) #得到矩阵 x^t y
x=np.append(x1,x2) #拼接 array
a=X[1] #取出矩阵的一行
a=X[:,1] #取出矩阵的一列
X[:,0]=a #更改矩阵的一列
a=X[n,m] #取出满足条件的元素
(n,m)=np.where(X>1) #查找满足条件的元素坐标，n 和 m 都是 array
l=np.argwhere(X>1) #n 为坐标组成的二维 array

```

5.2.3 矩阵运算

Listing 5.4: calculation of matrix

```
c = np.dot(x,y) #矩阵乘法
c = x*y #对应元素相乘
c = np.transpose(a);c=a.T #矩阵转置
c = np.transpose(a, (2, 0, 1)) #调整矩阵轴的顺序
c = np.linalg.norm(x) #求 x 的 L_2 范数
result=np.linalg.inv(a) #求逆矩阵
result=np.linalg.det(a) #求行列式
result=np.linalg.matrix_rank(a) #求矩阵的秩
result = np.dot(a,np.linalg.inv(b)) #矩阵右除
result = np.dot(np.linalg.inv(a),(b)) #矩阵左除
matrix.sum(axis=0) #求和，axis 的轴会消失

```

5.3 Pandas

5.3.1 Series

Series 类型是一维数组，由 index 和 value 组成。

Listing 5.5: Series in Pandas

```
import pandas as pd

data=['A','B','C']; index=['a','b','c']
dict_data = {'a': 'A', 'b': 'B', 'c': 'C'}

series = pd.Series(data) #默认 index 从 0 开始编号
series_with_index = pd.Series(data,index=index) #规定 index
series_with_index = pd.Series(dict_data) #这一句与上一句相同
print(series.index,series.value) #会输出数组
print(series_with_index['a']) #调用 Series 的元素

```

5.3.2 DataFrame

DataFrame 的每列的名称为键，每个键对应一个数组，这个数组为值。

Listing 5.6: Create a dataframe

```
import pandas as pd

data = {'a':[1,2,3,4,5],'b':[6,7,8,9,10],'c':[11,12,13,14,15]}
index = ['A','B','C','D','E']
data_frame = pd.DataFrame(data) #创建 DataFrame 对象, 默认 index 从 0 开始编号
data_frame = pd.DataFrame(data, index=index) #规定 index
data_frame = pd.DataFrame(data, columns=['a','b']) #指定列
print(data_frame)

df.set_index('a',  inplace=True) #将 column `a' 设定成 index, 仅用于创建时没有默认 index 的情况
df.reset_index(inplace=True) #将 index 重置为从 0 开始编号, 仅用于创建时没有默认 index 的情况

```

5.3.3 读写数据

Pandas 模块可以将 csv 或 excel 文件转为 DataFrame 变量, 也可以将 DataFrame 变量写入 csv 或 excel 文件。

Listing 5.7: Read and write files using Pandas

```
import pandas as pd

data = pd.read_csv('data.csv')
data = pd.read_excel('data.xlsx')
data = pd.read_csv('data.csv', index_col='Id') #设置指定 column 作为 index
data = pd.read_csv('data.csv', na_values=['NA', 'Missing']) #将给定值替换成 NaN
d_parser = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %I-%p')
data = pd.read_csv('data.csv', parse_date=['Date'], date_parser=d_parser)
    #将 Date 列转为 datetime
data = pd.read_json('data.json', orient='records', lines=True) #读取列表形式的 json 文件

data.to_csv('data.csv', columns=['A','B'], index=False) #不写入行索引
data.to_csv('data.tsv', sep='\t') #写入 tsv 文件
data.to_excel('data.xlsx', columns=['A','B'], index=False) #写入 excel 文件
data.to_json('data.json', orient='records', lines=True) #) 以列表形式写入 json 文件,默认是字典形
式

```

5.3.4 数据库交互

pandas 中的数据也可以通过 SQL alchemy 进行快速的存储和访问。

Listing 5.8: pandas & database

```
from sqlalchemy import create_engine
```

```
import psycopg2

engine = create_engine('postgresql://dbuser:dbpass@localhost:5432/sample_db')

```

5.3.5 基本操作

Listing 5.9: Basic functions about DataFrame

```
A = data_frame.to_numpy(); data_frame = pd.DataFrame(A) #DataFrame 和 numpy 转换

pd.set_option('display.max_columns', 20) #设置最多显示列数
pd.set_option('display.max_rows', 20) #设置最多显示行数
print(data_frame.columns) #输出所有列
print(data_frame.shape) #输出行列数
print(data_frame.dtype) #输出每一列的数据类型，混合数据类型显示为 object
print(data_frame.info()) #输出 DataFrame 的基本信息
print(df.head(10)); print(df.tail(10)) #输出前/后 10 行

data_frame[['a', 'b']] #返回指定列组成的 DataFrame
data_frame.iloc[0] #输出第一行
data_frame.iloc[[0, 1]] #输出指定行
data_frame.iloc[[0, 1], [1, 2]] #输出第 1，2 行第 2，3 列
data_frame.loc[[0, 1], ['a', 'b']] #输出 index 为 0，1，column 为 `a',`b' 的 DataFrame
data_frame.loc[0:1, 'a':'b']] #与上面一句相同
    #Loc 的变量也可以是 filter(True/False Series)
data_frame[data_frame > 0] #Filtering，用 NaN 对齐
data_frame[data_frame['b'].isin([7, 8, 9])] #Filtering using list
data_frame[data_frame['b'].str.contains('Python', na=False)]
    #提取 column `b' 字符串中包含 `Python' 的行，NaN 不提取

#updata column
data_frame.columns = ['Column_'+i for i in data_frame.columns]
data_frame.columns = df.columns.str.replace(' ', '_') #将 column 中的空格替换为下划线
data_frame.rename(columns={'a': 'Column_a', 'b': 'Column_b'}, inplace=True) #替换 columns 名

#update values
data_frame.loc['A':'B', 'a'] = [10, 11] #更新若干单元格
data_frame.loc['A':'B', 'a':'b'] = [[10, 11], [12, 13]] #更新若干单元格
data_frame.loc['B'] = [11, 12, 13] #更新一行
data_frame['d'] = ['AXIS', 'Root', 'TeX', 'Python', 'JavaScript'] #增添或修改一列数据
data_frame['d'] = data_frame['d'].str.lower() #全部改成小写
data_frame._append({'a': 17}, ignore_index=True) #插入一行，reset index 以避免冲突，用 NaN 补齐
df_total = data_frame._append(df, ignore_index=True) #合并两个 DataFrame
df_total = data_frame.concat([df1, df2], axis='columns') #合并两个 DataFrame
df_total = data_frame.concat([df1, df2], axis='index') #合并两个 DataFrame

#drop values
data_frame.drop(index=['A', 'B'], inplace=True) #按 index 删除，inplace 表示直接替换原数据
data_frame.drop(columns=['a', 'b'], inplace=True) #按 column 删除

```

5.3.6 统计操作

Listing 5.10: Perform statistical operations using DataFrame

```
#数据预处理
data_frame['d_len'] = data_frame['d'].apply(len) #对每个元素进行函数操作
data_frame.apply(len, axis='rows') #对每一行进行 len 操作，即获取 column 数
df_len = data_frame.applymap(len) #对 DataFrame 中每个元素进行 len 操作
sub_d = data_frame['d'].map({'Python': 'C++', 'JavaScript': 'Java'})
    #替换 Series 的内容，未匹配的替换成 NaN
sub_d = data_frame['d'].replace({'Python': 'C++', 'JavaScript': 'Java'})
    #替换 Series 的内容，未匹配的会保持原来的值

#处理空缺值
```

```
        null_num = data_frame.isnull().sum() #统计空缺值数量，isnull 在空缺值返回 True，否则返回 False
        not_null_num = data_frame.nornull().sum() #统计非空缺值数量
        data_frame.dropna(axis='index', inplace=True) #删除包含空缺值的整行数据
        data_frame.dropna(axis='index', how='all') #删除整行都空缺的数据
        data_frame.dropna(axis='index', how='any', subset=['name', 'age']) #删除指定列有空缺的行
        data_frame.fillna(0, inplace=True) #修改空缺值
        data_frame.fillna(data_frame.mean()) #平均值填充
        data_frame.fillna({'A':0,'B':1,'C':2},inplace=True) #每列空缺值用指定的值代替

        #处理不同数据类型
        data_frame['age'] = data_frame['age'].astype(float) #全部转换成 float
        data_frame['age'].unique() #返回所有出现的值

        #基本运算（逐项运算）
        score = data_frame.a + data_frame.b - data_frame.c #可以直接做向量运算
        filt = data_frame > 0

        #字符串操作（逐项操作）
        df['school'] = df['school'].str.upper()
        df['name'] = df['name'].str.split(' ')
        df[['first_name', 'last_name']] = df['name'].str.split(' ', expand=True)
        df['name'].str.contains('Jack') #匹配字符串，返回一个 True-False Series

        #常用的统计函数
        data_frame.sort_index(inplace=True) #按 index 排序，NaN 会被当成最大值
        data_frame.sort_values(by='a', ascending=False, inplace=True) #排序，ascending 为升序
        data_frame.sort_values(by=['a', 'b'], ascending=[True, False]) #第一标准相同排第二标准，以此类推
        #以下函数在运行时会忽略 NaN
        data_frame.nlargest(5, 'a') #取 `a' 最大的五行
        data_frame.nlargest(5, columns=['a', 'b']) #取最大五项，第一标准为 `a'，第二标准为 `b'
        data_frame.nsmallest(5, columns=['a', 'b']) #取最小五项，第一标准为 `a'，第二标准为 `b'
        nums = df['a'].count() #统计非 NaN 的个数
        nums = df['a'].value_counts() #统计 values 出现的频次，返回一个 Series
        nums = df['a'].value_counts(normalize=True) #统计 values 出现的频率
        average = data_frame.mean() #求每列的平均值，输出一个 Series
        media = data_frame.median() #求每列的中位数
        sum = data_frame.sum() #求每列的和
        variance = data_frame.var() #样本方差
        std = data_frame.std() #样本标准差
        print(data_frame.describe()) #给出数据的所有常见统计参数

        #分组
        country_grp = df.groupby(['Country'])
        print(country_grp.get_group('United States')) #查找一个 group，返回一个 DataFrame
        country_grp['SocialMedia'].value_counts().head() #统计每一个 group 的 SocialMedia，返回 Series
        country_grp['Salary'].agg(['median', 'mean']) #返回 DataFrame，columns=['median', 'mean']
        country_grp['Language'].apply(lambda x: x.str.contains('French').sum())#将函数作用于每个 group

```

5.3.7 处理 datetime

Listing 5.11: datetime in pandas

```
        d_range = pd.date_range('2024-01-01', period=6, freq='D') #区间生成日期 Series
        df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d %I-%p') #string 转 datetime

        df['Date'].dt.date() #对整行 datetime 进行操作
        df['Date'].max(); df['Date'].min() #获取最早和最晚的时间
        df.loc[df['Date'] >= pd.to_datetime('2020-5-20')]

        df.set_index('Date', inplace=True) #下面将 Date 设置为 index
        df['2019'] #得到 2019 年的数据
        df['2019-02':'2020-01'] #得到指定时间区间内的数据
        df['temperature'].resample('2D').max()
            #将数据重采样为每两天的频率并求最大值，重采样方式参考date offset Series
        df.resample('W').agg({'temperature': 'max', 'rain': 'sum', 'air_quality': 'mean'})
            #重采样时，对不同 column 执行不同的操作

```

6 visualization

6.1 matplotlib

6.1.1 画布预处理

Listing 6.1: basic setup of plt

```
from matplotlib import pyplot as plt

plt.figure(figsize=(10,20),facecolor,edgecolor)
plt.title("This is my title")
plt.xlabel("x"); plt.ylabel("y")
plt.xlim(2,22) #设置 x 的范围
plt.xscale('log') #设置 x 为对数轴
plt.xticks(ticks=[2*i+1 for i in range(10)],
           labels=[2*i+1 for i in range(10)]) #设置坐标轴位置和标签
plt.legend(["line1", "line2"], loc="upper left") #显示图例，以加入图表的顺序为序
plt.legend(loc=(0.03, 0.05)) #显示图例，利用图表绘制时的 label，直接以横纵坐标设定位置
plt.rcParams['font.sans-serif'] = ['SimHei'] #字体设置 (默认字体不支持中文)
plt.rcParams['axes.unicode_minus'] = False #正常显示负号

plt.style.use("seaborn-v0_8") #利用预设样式
plt.xkcd() #手绘样式
plt.tight_layout() #布局更紧凑，可以避免 label 和 ticks 显示在画布范围外的问题

plt.grid(True) #开启网格线
plt.grid(axis=both, #axis=x or y or False
        linestyle="dashed", #or dotted or dashdot
        color="#FFFFFF"
) #添加网格线
plt.axhline(5, #水平参考线
            xmin=0.1, xmax=0.9, #[0, 1] 范围内 i 昂对于 x 轴的归一化范围
            color="red", linestyle="--", linewidth=0.5 #参考线样式
)
plt.axvline(10) #垂直参考线
plt.axhspan(5, 7, #水平参考区域
            xmin=0.1, xmax=0.9, #[0, 1] 范围内 i 昂对于 x 轴的归一化范围
            color="red", alpha=0.5, #填充样式，color 会同时作用于 facecolor 和 edgecolor
            linestyle="--", linewidth=0.5 #区域边沿线样式
)
plt.axvspan(10, 12) #垂直参考区域

plt.annotate(text,xy=(5,10), #待注释点坐标
            xytext=(7,12), #注释文本位置
            color="#FFFFFF",fontsize=16,
            ha="center", #水平居中
            va="bottom", #垂直对齐
            arrowprops={"arrowstyle":"->", #or "-"
                        "color":"#FFFFFF"}
) #显示注释点
plt.text(7,12,text) #显示无箭头注释

plt.savefig('plot.png') #保存图片
plt.show() #显示图片，注意没有这句不会显示

```

6.1.2 子图表

figure 是一整张图片，plot 是其中的图。一个 figure 可以有多个 plot。

Listing 6.2: subplot

```
ax = plt.gca() #返回当前坐标轴
fig = plt.gcf() #返回当前 figure

fig, ax = plt.subplots( #创建 figure 和其中的 subplot，ax fig 是 list
    nrows=2, ncols=1,
```

```
        sharex=True, #共用 x 轴 (xticks 只保留一个, xlabel 不会共用)
    )

    ax[0].plot(x, y)
    ax[1].plot(x, y1)
    ax[0].legend()

    ax[0].set_title("Title of plot")
    ax[0].set_xlabel("x")
    ax[0].set_ylabel("y")

    plt.tight_layout()
    plt.show()
    fig.savefig('test.jpg')

```

6.1.3 数据获取

Listing 6.3: fetch data

```
    import numpy as np
    import csv

    x = np.arange(5)
    y = np.array([3, 7, 8, 9, 13])
    y2 = np.array([2, 4, 7, 11, 16])

    with open('data.csv') as csv_file:
        csv_reader = csv.DictReader(csv_file)
        x = csv_reader.fieldnames
        y = [float(i) for i in next(csv_reader).value()]
        y1 = [float(i) for i in next(csv_reader).value()]

```

6.1.4 折线图

Listing 6.4: line plot

```
    plt.plot(x, y, 'k--') #黑色虚线, 第三个变量是 format strings 样式, 见文档 Notes part
    plt.plot(x, y,
            color="#5a7d9a",
            label="Line1",
            linestyle="dashed", #or dotted or dashdot
                #也可以设置为-- -. - : None '' ' '
            linewidth=3,
            marker=".", #or "," "o" "+" "x"
            markersize=8,
            markerfacecolor="blue",
            markeredgecolor="cyan",
            markeredgewidth=2
    )
    #面积图, y 轴是各组 y 数据的和
    plt.stackplot(x, y1, y2, y3,
                  color=["red","yellow","blue"],
                  labels=["player1", "player2", "player3"]
    )
    plt.fill_between(x, y, alpha=0.25) #填充 x-y 下方的区域
    plt.fill_between(x, y, 30, #填充 x-y 和 y=30 之间的区域
                     where=(y > 30), #仅填充 y>30 的部分
                     interpolate=True,
    )
    plt.fill_between(x, y, y1) #填充两条曲线之间的部分

```

6.1.5 柱状图

Listing 6.5: bar chart

```
        plt.bar(x, y,
                width=0.8, #柱形宽度
                bottom=1, #柱形底部高度，可以是和 x 一样长的列表，用于 bar 的堆叠
                hatch="/") #or "L" ""\\" "//"
        plt.bar(['Jack', 'Ron', 'Jane'], [93, 98, 96]) #这里 x 轴可以不是数
        plt.barh(x,y) #横向柱状图，适用于 bar 较多的情况

        width = 0.25 #这是并列柱状图的画法
        plt.bar(x-width/2, y, width=width, color="#008fd5", label="y")
        plt.bar(x+width/2, y1, width=width, color="#e5ae38", label="y1")
        #直方图
        plt.hist(x, bins=5, #直方图，bins 可以是整数 (分组数量) 或列表
                 edgecolor="black"
        )
        plt.hist(x, bins=[10, 20, 30, 40]) #列表设定各 bins 的边界
        plt.hist(x, bins=5,
                 log=True #y 轴以 10 的幂次表示，用于数据差距比较大的情况
        )

```

6.1.6 饼图

Listing 6.6: pie chart

```
        #普通饼图
        plt.pie(x, #不用换算成百分数，原数据即可
                labels=["一月","二月","三月"],
                colors=["red","blue","yellow"],
                wedgeprops={"edgecolor":"red"} #各区域之间的分割线
                autopct="%0.1f%%", #一位小数百分比，%d%% 整数百分比，%0.1f 一位小数，%0.2f%% 两位小数百分比
                explode=[0,0.1,0], #将第二块拉出 0.5
                shadow=True, #加上阴影
                startangle=90, #开始方向，默认为右侧 (0)，逆时针为正方向
        )
        #圆环图
        plt.pie(x,colors=["red","blue","yellow"],
                shadow=True,labels=["一月","二月","三月"],
                wedgeprops={"width":0.6} #内圆半径，外圆半径为 1
        )

```

6.1.7 散点图

Listing 6.7: scatter plot

```
        plt.scatter(x, y, s=100, #散点图，s 表示点的大小，可以是和 x 一样长的 list
                    color="red", alpha=0.5, #填充样式
                    marker="0", #or "," "+" "x" "^"
                    edgecolor="black", linewidth=2, linestyle="-.", #边缘线样式
        )
        plt.scatter(x, y, [i*50+100 for i in range(len(x))]) #气泡图

        plt.scatter(x, y, c=[7, 5, 9, 2, 3], cmap='Greens') #通过 cmap 设置颜色
        cbar = plt.colorbar(label='heat') #显示 colorbar 及其注释
        #可以通过 print(plt.colormaps) 查看所有的可选 cmap
        #误差棒图
        plt.errorbar(x, y, #散点数组
                    yerr=[[0.1, 0.3, 0.2], [0.4, 0.1, 0.1]], #上置信度和置信度
                    xerr=[0.05, 0.04, 0.07], #上下置信度
                    fmt="o--", #散点样式和连线样式
                    color="red", #color of scatters
```

```
                    ecolor="blue", #color of the errorbars
                    elinewidth=3, #误差棒粗细
                    capsize=2, #误差棒横杠大小
                    capthick=2, #误差棒边界横杠厚度
                    uplims=False, #显示误差上界，False 显示，True 隐藏
                    lolims=[True, False, False], #只显示指定的下界
        ) #其余选项和 plot,scatter 类似

```

6.1.8 箱型图

Listing 6.8: box plot

```
        #箱型图
        plt.boxplot(x, #x 是一个数组
                    showmeans=True, #显示均值
                    patch=True, #显示缺口箱图，默认矩形箱图
                    vert=False, #False 为横向，True 为纵向
                    showfliers=True, #显示离群点，默认显示
                    whis=1.5, #箱须延长的百分比范围
                    width=1, #箱体总宽度
                    flierprops={ #离群点样式
                        "marker":"o", #or "," "+" "x"
                        "markerfacecolor":"red",
                        "markeredgecolor":"black",
                        "markersize":8
                    },
                    patch_artist=True, #自定义箱型
                    boxprops={ #箱型样式
                        "facecolor":"red",
                        "edgecolor":"yellow",
                        "linewidth":4
                    },
        )
        #绘制多个箱型
        plt.boxplot((x, y), #适用于 x 轴没有数值含义的时候
                    tick_labels=['x', 'y'], #均匀分布
        )
        plt.boxplot((x, y), #适用于 x 轴有数值
                    positions=(1, 2),
                    manage_ticks-True #刻度会根据 positions 自动调整
        )

```

6.1.9 热力图

Listing 6.9: heat map

```
        #绘制热力图
        plt.imshow(x, #x 是个二维列表
                   cmap=plt.cm.cool #设置颜色
        )
        plt.colorbar(label="label") #显示图例

```

6.1.10 时间图像

Listing 6.10: Plotting Time Series Data

```
        import pandas as pd
        from datetime import datetime, timedelta
        from matplotlib import pyplot as plt
        from matplotlib import dates as mpl_dates
```

```
#获取时间
dates = pd.Series([datetime(2024, 6, 30), datetime(2024, 7, 5), datetime(2024, 7, 11)])
dates = pd.Series(["2024-6-30", "2024-7-5", "2024-7-11"])
dates = pd.to_datetime(dates).sort_values() #注意需要排序
y = [4, 6, 7]

plt.plot(dates, y)
plt.gcf().autofmt_xdate() #将日期斜方向写，gcf 表示当前图像

date_format = mpl_dates.DateFormatter('%b, %d %Y')
plt.gca().xaxis.set_major_formatter(date_format) #设置日期格式

```

6.1.11 极坐标图表

Listing 6.11: polar plots of plt

```
plt.polar(theta,r) #雷达图
plt.thetagrid(angles,labels) #角刻度标签
plt.rgids(radii,rotation,labels) #r 方向刻度标签

ax=plt.axes(polar=True) #建立极坐标画布
ax.bar(x=theta,height=data,width=0.4,color="rainbow") #绘制南丁格玫瑰图
ax.bar(x=theta,height=100,width=0.4,color="white") #绘制中心空白
ax.text(angle,height,text) #添加注释
ax,grid(False)
plt.thetagrids(angles=[],labels=[]) #刻度标签
plt.rgrids(radii=[20],rotation,labels=['20'])

```

6.1.12 三维图表

Listing 6.12: 3D plots of plt

```
from mpl_tookits.mplot3d import Axes3D
fig=plt.figure()
ax1=plt.axes(projection="3d")
ax1.scatter3D(x,y,z,cmap="blue")
ax1.plot3D(x,y,z,"gray")
ax1.plot_surface(X,Y,Z,rstride=0.1,cstride=0.1) #步长越短越清晰
ax1.contour(X,Y,Z,zdir='x',offset=-3,cmap="cold") #绘制等高线，投影在 x=3 平面上
ax1.bar3d(X,Y,height,width,depth,Z,color="red",shade=True) #绘制柱状图,height 为柱底高度

```

6.1.13 动态实时图像

Listing 6.13: plotting live date

```
from matplotlib.animation import FuncAnimation
import itertools, random

x_vals, y_vals, index = [], [], itertools.count()
def animate(i):
    x_vals.append(next(index))
    y_vals.append(random.randint(0, 5))
    plt.cla() #先清除上一帧图像
    plt.plot(x_vals, y_vals)
    plt.tight_layout()

ani = FuncAnimation(plt.gcf(), animate, interval=1000)

plt.tight_layout()
```

```
        plt.show()

```

6.2 wordcloud

Listing 6.14: wordcloud

```
import matplotlib.pyplot as plt
import wordcloud as wc

text_data = """
Python is a popular programming language.
It is widely used for web development, data analysis, and artificial intelligence.
Word clouds are fun visualizations of text data.
Generate a word cloud using the wordcloud module.
"""

# 生成词云对象
wordcloud = wc.WordCloud(width=800, height=400, background_color='white').generate(text_data)

# 显示词云图
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()

```

7 website

7.1 请求连接

Python 自带的 urllib 和 urllib3 模块可以实现一些网络爬虫的常用操作。外接库中的 requests 库则更为常用。

7.1.1 urllib module

利用 request 模块可以实现 get 请求方式获取网页内容。

Listing 7.1: Fetch web content using GET method

```
import urllib.request #网络请求子模块

response = urllib.request.urlopen('http://www.baidu.com') #打开网页
html = response.read() #读取网页代码
response.close()

#也可以用 with 语句简化
with urllib.request.urlopen('http://www.baidu.com') as response:
    html = response.read()
print(html)

```

也可以实现 post 请求方式获取网页内容。

Listing 7.2: Fetch web content using POST method

```
import urllib.parse #url 解析和引用模块
import urllib.request

#使用 urlencode 方法对数据进行处理，并将处理后的数据设置为 utf-8 编码
data = bytes(urllib.parse.urlencode({'word':'hello'}), encoding='utf8')
with urllib.request.urlopen('http://httpbin.org/post', data=data) as response#打开网页
    html = response.read()
```

```
        print(html)
```

7.1.2 urllib3 module

urllib3 是一个更强大的 Python 库。

Listing 7.3: urllib3 module

```
        import urllib3

        #用 GET 的方式连接
        http = urllib3.PoolManager() #创建对象，用于处理连接和安全等细节
        response = http.request('GET','https://www.baidu.com/') #连接网站
        print(response.data) #输出读取内容

        #用 POST 的方式连接
        http = urllib3.PoolManager() #创建对象，用于处理连接和安全等细节
        response = http.request('POST', 'http://httpbin.org/post', fields={'word':'hello'})
        print(response.data) #输出读取内容
```

7.1.3 requests module

另外有个更加人性化的第三方库 requests，这个库更加常用。

Listing 7.4: requests module

```
        import requests

        data={'word':'hello'}
        response = requests.get('http://www.baidu.com',params=data) #get 方法访问
        response = requests.post('http://httpbin.org/post',data=data) #post 方法访问
        if response.ok: #检验连接是否成功
            print(response.content) #以字节流形式输出网页源码（没有换行符和缩进，可读性差）
            print(response.text) #以文本形式输出网页源码（带有换行符和缩进，可读性好）

        #下载文件
        import zipfile
        r = requests.get('<url>') #zip file 的 url
        with open('data.zip', 'wb') as f: #建一个 data.zip 用于保存
            f.write(r.content)
```

为了绕开反爬设计，我们可以请求 header。

Listing 7.5: Handling request headers

```
        import requests

        url = 'http://www.bilibili.com/'
        headers = {'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
            (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36'}
        #这里 User-Agent 的内容要从网络监视器中复制过来
        response = requests.get(url,headers=headers)
        print(response.content)
```

7.2 处理 html 文件

BeautifulSoup module 用于处理 html 文件

Listing 7.6: Handling HTML

```
from bs4 import BeautifulSoup

with open('test.html') as html_file: #打开本地的 html 文件
    soup = BeautifulSoup(html_file, 'lxml') #用 lxml 解析器解析 html 文件

source = requests.get('http://bilibili.com').text
soup = BeautifulSoup(source, 'lxml') #从网页上获取文件

print(soup.prettify()) #转换成易读的 html 文件
match = soup.title.text #得到文件中的某个部分
match_div = soup.find('div', class_='footer') #查找文件中第一个对应的内容
match_all = soup.find_all('div') #查找文件中所有对应的内容，返回一个列表

```

7.3 Flask

Flask 是一个常见的网站开发框架。

7.3.1 create a website

Listing 7.7: create a web site

```
#my_app.py
from flask import Flask, abort
app = Flask(__name__)

@app.route("/") #root page
@app.route("/home") #两个网址共用一个网页
def hello():
    return "<h1>Hello World!</h1>"

@app.route("/error") #两个网址共用一个网页
def hello():
    abort(403) #返回 403 错误 (forbidden)

if __name__ == '__main__':
    app.run(debug=True)

```

Listing 7.8: start a web site

```
export FLASK_APP=my_app.py
flask run #运行网站，每次修改 python 文件需要重新启动
export FLASK_DEBUG=1
flask run #运行网站，每次修改都会立即出现在网站上

python my_app.py #直接运行网站

```

7.3.2 template

template 就是每个 url 的 html 模板，建立 template 可以避免将 html 语言写到 python 文件中造成混乱。

在下面的例子中，我们将为 root page 创建一个 template/home.html 文件，注意这个文件必须放在 template 文件夹下。

Listing 7.9: template/home.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8"> <!--提供文档的元数据-->
```

```
            <title>网页标题</title>
            <style> <!--定义内部CSS样式-->
                .custom-text {
                    font-family: "Times New Roman", serif;
                    color: green;
                }
            </style> <!--更好的做法是将要用到的style都写到一个.css文件中
        </head>
        <body>
            <header>
                <h1>欢迎来到我的网站</h1>
                <nav>
                    <ul>
                        <li><a href="{{ url_for('home') }}">主页</a></li>
                        <li><a href="#about">关于我们</a></li>
                        <li><a href="/contact">联系我们</a></li>
                    </ul>
                </nav>
            </header>
            <main> <!--main part of body-->
                <h1>这是一个<span style="color:red;">标题<\span></h1> <!--h1至h6表示标题的等级-->
                <p style="custom-text">这是一个段落。</p></br> <!--br为空格-->

                <strong>这是重要的内容。</strong>
                <small>这是一个较小的文本。</small>
                <em>这是一个需要强调的文本。</em>
                <mark>这是一个被标记的文本。</mark>
                <p>H<sub>2</sub>O 表示水。</p> <!--下标文本-->
                <p>2<sup>2</sup> = 4</p> <!--上标文本-->
                <p>这是一个 <code>print("Hello, World!")</code> 示例。</p>

                <a href="https://www.example.com">访问示例网站</a>
                <a href="#">点击这里回到顶部</a>
                <img src="image.jpg" alt="描述文本">
            </main>

            <h2>无序列表</h2>
            <ul> <!--ol有序无标记列表，li有序有标记列表-->
                <li>苹果</li>
                <li>香蕉</li>
                <li>橙子</li>
            </ul>

            <div style="border: 1px solid black; padding: 10px;">
                <h2>新闻标题</h2>
                <p style="font-family: Arial, sans-serif; color: blue;">
                    这是新闻的内容部分。该部分使用`&lt;div&rt;`标签包裹，以便进行样式处理和布局控制。
                </p>
            </div>
            <div>
                <button>Click me!</button> <!--按钮-->
                <input type="text" name="username" placeholder="Enter name"> <!--文本输入框-->
                <input type="password" name="password" placeholder="Password"> <!--密码输入框-->
                <input type="submit" value="Submit"> <!--提交按钮-->
                <input type="radio" name="gender" value="male"> <!--单选按钮-->
                <input type="radio" name="gender" value="female"> <!--单选按钮-->
                <input type="checkbox" name="agree" value="yes"> <!--复选框-->
            </div>
        </body>
        </html>

```

html 文件中有一些特殊字符需要转义:

- 大于号小于号 < >: < &rt;
- 书名号《 》: ≪ &Rt;
- 与号 &: &
- 双引号”: "

- 单引号': '
- 空格":
- 乘号 ×: ×
- 除号 ÷: ÷

Listing 7.10: using template in my_app.py

```
@app.route("/")
def home():
    return render_template('home.html')

@app.route("/post/<int:post_id>") #route 可以带有变量，这里设置其为 int 类型
def post(post_id):
    post = Post.query.get_or_404(post_id) #如果网页存在就返回 post_id，否则返回 404
    return render_template('post.html', post=post)

```

7.3.3 template inheritance

同一个网站中 template 往往有很多重合部分，template inheritance 可以实现统一设置模板。下面，我们先创建一个 layout.html，其中包含网页的公共部分。

Listing 7.11: layout.html

```
<head>
    {% if title %}
        <title>My app - {{ title }}</title>
    {% else %}
        <title>My app</title>
    {% endif %}
</head>
<body>
    {% block content %}{% endblock %} <!--需要替换的部分-->
<\body>

```

在 home.html 中使用 layout.html。

Listing 7.12: Using template inheritance

```
{% extends "layout.html" %}
{% block content %}
  {% for post i posts %} #读出posts的内容
      <h1>{{ post.name }}</h1> #两个大括号表示代码结果要输出到网页上
      <p>with number of {{ post.number }}</p>
  {% endfor %} #结束读取
{% endblock content %}

```

7.3.4 bootstrap

bootstrap 定义了一些方法来优化网页，参考bootstrap 官方文档。

Listing 7.13: bootstrap

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
```

```
                rel="stylesheet"
                integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH"
                crossorigin="anonymous">

            {% if title %}
                <title>My app - {{ title }}</title>
            {% else %}
                <title>My app</title>
            {% endif %}
        </head>
        <body>
            <div class="container" mt-1>  <!--利用bootstrap中的设定-->
                <!--外间距:mt上间距，mb下间距，my上下间距-->
                <!--外间距:ml左间距，mr右间距，mx左右间距-->
                <!--内间距:pt左间距，pr右间距，px左右间距-->
                <!--内间距:pl左间距，pr右间距，px左右间距-->
                {% block content %}{% endblock %} <!--需要替换的部分-->
            </div>

            <script
                src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
                integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz"
                crossorigin="anonymous">
            </script>
        </body>
        </html>

```

7.3.5 External CSS

External CSS(外部样式表) 将网页中文本、图片等元素所需样式都写在 static/main.css 文件里备用。main.css 中可以为每个板块类定义样式，这个样式会自动生效于网页中所有对应的类。

下面这个样例表示在 'article-metadata' 类的元素内的链接 (<a> 标签) 当鼠标悬停 (hover) 时的样式。

Listing 7.14: static/main.css

```
        .article-metadata a:hover {
          color: #333;
          text-decoration: none;
        }

```

导入 External CSS 需要用到 url_for module

Listing 7.15: url_for

```
        from flask import Flask, render_template, url_for

        url_for('home') #返回 home 函数对应的 url
        url_for('static', filename='picture1.jpg') #在 static 文件夹下 picture1.jpg 的地址
        url_for('post', post_id=1) #返回 url:  /post/1, 用于 decorator 里有变量的情况

```

在 layout.html 中调用 url_for 导入样式表。

Listing 7.16: layout.html

```
        <head>
            <link rel="stylesheet" type="text/css"
                  href="{{ url_for('static', filename='main.css') }}">
            <!--static为默认目录，main.css只要在这个目录下面，url_for都会自动找到该文件-->
        </head>

```

7.3.6 post data

template 中可以通过 post 传递变量，以实现信息交换。

Listing 7.17: post data in flask

```
        posts = [ #posts 是一个 dict 组成的 list
            {'name': 'Amy', 'number': 11},
            {'name': 'Jack', 'number': 28}
        ]

        def home():
            return render_template('home.html', posts=posts, title="Home page")

```

Listing 7.18: use posts in html

```
        <head>
            {% if title %}
                <title>My app - {{ title }}</title>
            {% else %}
                <title>My app</title>
            {% endif %}
        </head>
        <body>
            {% for post i posts %} #读出posts的内容
                <h1>{{ post.name }}</h1> #两个大括号表示代码结果要输出到网页上
                <p>with number of {{ post.number }}</p>
            {% endfor %} #结束读取
        <\body>

```

7.3.7 form & field

form 表示网页上的表单，field 表示提供用户输入的区域。一个 form 可以有多个输入。flask-wtf module 可用于处理表单。其中 wtforms 提供了各种表单区域，详见wtf field。wtforms.validators 提供了检验输入法的方法，详见wtf validators。

下面，我们建立一个 forms.py 文件来处理表单

Listing 7.19: form.py

```
        from flask_wtf import FlaskForm
        from wtforms import StringField, PasswordField, SubmitField, BooleanField
            #这里常用的还有还有 TextAreaField 文本编辑区域
        from wtforms.validators import DataRequired, Length, Email, EqualTo, ValidationError
        from models import User #models 将在下一目建立

        class RegistrationForm(FlaskForm):
            username = StringField('Username', validators=[DataRequired(), Length=(min=2, max=20)])
            email = StringField('Email', #这里的第一个变量就是下面 html 文件中的 label
                                validators=[DataRequired(), Email()])
            password = PasswordField('Password', validators=[DataRequired()])
            confirm_password = PasswordField('Confirm Password',
                                             validators=[DataRequired(), EqualTo('password')])
            remember = BooleanField('Remember Me')
            submit = SubmitField('Sign Up')

            def validate_username(self, username): #Validate 函数会在表单提交时自动调用
                user = User.query.filter_by(username=username.data).first()
                if user: #用户名已存在
                    raise ValidationError('That username is taken.')

            def validate_email(self, email):
                user = User.query.filter_by(email=email.data).first()
                if user: #用户名已存在
                    raise ValidationError('That email is taken.')

```

创建 register.html 显示表单。<form> 用于建立表单，<fieldset> 用于将表单中的内容进行分组。

Listing 7.20: register.html

```
{% extends "layout.html" %}
{% block content %}
    <div class="content-section">
        <form method="POST" action=""> <!--action=""表示将表单数据提交到当前URL-->
            {{ form.hideen_tag() }} <!--防止跨站请求伪造攻击-->
            <fieldset class="form-group">
                <legend class="border-bottom mb-4">Join Today</legend>
                <!--Legend 一般用于fieldset内部，作为注释-->
                <div class="form-group">
                    {{ form.username.label(class="form-control-label") }}
                    {% if form.username.errors %}
                        {{
                            form.username(
                                class="form-control form-control-lg is invalid"
                            )
                        }}
                        <div class="invalid-feedback">
                            {% for errors in form.username.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.username(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.email.label(class="form-control-label") }}
                    {% if form.email.errors %}
                        {{
                            form.email(
                                class="form-control form-control-lg is invalid"
                            )
                        }}
                        <div class="invalid-feedback">
                            {% for errors in form.email.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.email(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.password.label(class="form-control-label") }}
                    {% if form.password.errors %}
                        {{
                            form.password(
                                class="form-control form-control-lg is invalid"
                            )
                        }}
                        <div class="invalid-feedback">
                            {% for errors in form.password.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.password(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.confirm_password.label(class="form-control-label") }}
                    {% if form.confirm_password.errors %}
                        {{
                            form.confirm_password(
                                class="form-control form-control-lg is invalid"
                            )
                        }}
```

```
                                <div class="invalid-feedback">
                                    {% for errors in form.confirm_password.errors %}
                                        <span>{{ error }}</span>
                                    {% endfor %}
                                </div>
                            {% else %}
                                {{ form.confirm_password(class="form-control form-control-lg") }}
                            {% endif %}
                        </div>
                    </fieldset>
                    <div class="form-group">
                        {{ form.submit(class="btn btn-outlne-info") }}
                    </div>
                </form>
            </div>
            <div class="border-top pt-3">
                <small class="text-muted">
                    Already Have An Account?
                    <a class="ml-2" href="{{ url_for('login') }}">Sign In</a>
                </small>
            </div>
        {% endblock content %}

```

在 app.py 中设置密码 (防止用户信息泄露)，调用表单，接收用户提交的表单，并在处理表单后重定向。

Listing 7.21: Using forms in main.py

```
        from forms import RegistrationForm
        from flask import Flask, render_template, url_For, redirect

        app.config['SECRET_KEY'] = '114514' #密码可以由 secrets module 生成

        @app.route("/register", methods=['GET', 'POST']) #接收 GET 和 POST 表单
        def register():
            form = RegistrationForm()
            if form.validate_on_submit(): #判断提交表单是否合理
                flash(f'Account created for {form.username.data}!', 'success') #显示一次性消息
                    #消息类别有"error""danger""warning""success" 等
                return redirect(url_for('home'))
            return render_template('register.html', title='Register', form=form)

```

在 layout.html 中显示 flash 的内容并完成重定向。

Listing 7.22: layout.html

```
        {% with messages = get_flashed_messages(with_categories=true) %} <!--success就是true-->
            {% if messages %}
                {% for category, message in messages %}
                    <div class="alert alert-{{ category }}">
                        {{ messages }}
                    </div>
                {% endfor %}
            {% endif %}
        {% endwith %}

```

7.3.8 flask-sqlalchemy

flask-sqlalchemy 可以用于存储网站运行产生的数据。
创建一个 models.py 文件，用于处理数据库。

Listing 7.23: Using sqlalchemy in models.py

```
        from flask_sqlalchemy import SQLAlchemy
```

```
from datetime import datetime
from app import db

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db' #///表示相对路径
db = SQLAlchemy(app)

class User(db.Model): #创建一个 User 表
    id = db.Column(db.Integer, primary_key=True) #不重复的 key，会自动生成，从 1 开始
    username = db.Column(db.String(20), unique=True, nullable=False) #用户名不重复且不为空
    email = db.Column(db.String(120), unique=True, nullable=False)
    image_file = db.Column(db.String(20), nullable=False, default='default.jpg')
        #图片用 hash 值记录
    password = db.Column(db.String(60), nullable=False) #密码用 hash 值记录
    posts = db.relationship('Post', backref='auther', lazy=True) #记录 Post 和 User 的关系
        #这里 Post 指代下面的 Post 类型，lazy=True 会使实例直接带有 relationship 信息

    def __repr__(self):
        return f"User('{self.username}', '{self.email]', '{self.image_file}')"

class Post(db.Model): #创建一个 Post 表
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    date_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
        #注意这里 utcnow 是一个函数而不是函数的返回值，否则 default 会立即调用，生成一个固定的时间
    content = db.Column(db.Text, nullable=False) #长文本类型
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
        #这里 user.id 指代 user 表中的 id 列 (表名是类名的小写)
        #表示 user_id 是 user 表 id 列的一个外键

    def __repr__(self):
        return f"Post('{self.title}', '{self.date_posted]')"

```

在 app.py 文件中使用 models.py

Listing 7.24: Using models in main.py

```
from models import User, Post

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db' #///表示相对路径
db = SQLAlchemy(app) #database 和 app 绑定，再次运行不会生成新的 database

```

在 shell 创建数据库文件，并尝试加入数据。

Listing 7.25: create database file in terminal

```
>>> from app import db
>>> db.create_all()

>>> from app import User, Post
>>> user_1 = User(username='Jack', email='J@demo.com', password='password')
>>> db.session.add(user_1)
>>> db.session.commit()

>>> User.query.all() #输出所有 User
>>> User.query.first() #输出第一个 User
>>> User.query.filter_by(username='Jack').all()
>>> user = User.quety.get(1) #按照 id 查找

>>> post_1 = Post(title='Blog 1', content='First Post Content!', user_id=user.id)
>>> db.session.add(post_1)
>>> db.session.commit()
>>> user.posts #此时可以输出用户的 posts
>>> db.query.first().author #利用 backref 查询作者

>>> db.drop_all() #删除数据库中的所有内容

```

将 website 产生的用户数据存入 database，这里用到了 hash 生成，见下一目。

```
@app.route("/register", methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        hash_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data,
                    email=form.email.data,
                    password=hashed_password)
        db.session.add(user)
        db.session.commit()
        flash(f'Account created for {form.username.data}! You can log in now', 'success')
        return redirect(url_for('login'))
    return render_template('register.html', title='Register', form=form)

```

7.3.9 Bcript & LoginManager

flask-bcrypt.Bcrypt 可用于生成 hash 值。flask_login.LoginManager 用于控制用户登录

Listing 7.26: Using Bcript in app.py

```
from flask import request #这将帮助我们得到 GET 的内容，以便重定向
from flask_bcrypt import Bcrypt
from flask_login import LoginManager, UserMixin, login_user,
                        current_user, logout_user, login_required
    #这些变量可以在 html 中直接访问，无需通过 render_template 传递

bcrypt = Bcrypt(); bcrypt.init_app(app)
bcrypt = Bcrypt(app) #上面两句是一样的
login_manager = LoginManager(app)
login_manager.login_view = 'login' #指向 login 函数
#当没有 login 而访问需要 login 的 url 时，会被重定向到 login，同时通过 GET 传递重定向前的 route
login_manager.login_message_catagory = 'info' #info 是一个内置类
#这一句会改变重定向时提示的样式

pwd_hash = bcrypt.generate_password_hash('password').decode('utf-8')
    #这个函数重复运行会生成不同的 hash 值
bcrypt.check_password_hash(pwd_hash, 'password') #返回 True
bcrypt.check_password_hash(pwd_hash, 'Testing') #返回 False

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int())

class User(db.Model, UserMixin):
    #defination of id etc.

@app.route("/login")
def login():
  user = User() #add name email etc.
  if current_user.is_authenticated:
      return redirect(url_for('home'))
  else:
      login_user(user) #user 会载入 current_user，后面可访问，如 current_user.username
      next_page = request.args.get('next') #得到重定向前的 route
      return redirect(next_page) if next_page else redirect(url_for('home'))

@app.route("/account")
@login_required #需要 login 才能访问
def account():
    form = UpdateAccontForm() #修改用户信息的表单
    if form.validate_on_submit():
        current_user.username = form.username.data #注意 form 要加 data 而 current_user 不用
        current_user.email = form.email.data
        db.session.commit() #直接修改 current_user 即可
        flash('Your account has been updated!', 'success')
        return redirect(url_for('account'))
    elif request.method == 'GET':
        form.username.data = current_user.username
```

```
            form.email.data = current_user.email
        image_file = url_for('static', filename='profile_pics/' + current_user.image_file)
        return render_template('account.html', form=form, image_file=image_file)

```

7.3.10 FileField

FileField 允许网页传输文件 (图片、文本文件等)。FileAllowed 设置了允许传输的文件类型。

Listing 7.27: FileField

```
from flask_wtf.file import FileField, FileAllowed
import secrets

def save_picture(form_picture):
    random_hex = secrets.token_hex(8) #生成随机文件名，防止同名文件 bug
    _, f_ext = os.path.splitext(form_picture.filename)
    picture_fn = random_hex + f_ext
    picture_path = os.path.join(app.root_path, 'static/profile_pics', picture_fn)
    form_picture.save(picture_path)
    return picture_fn

class UpdateAccountForm(FlaskForm):
    picture = FileField('Update Profile Picture',
                        validators=[FileAllowed(['jpg', 'png'])])

@app.route("/account", methods=['GET', 'POST'])
    if form.validate_on_submit():
        if form.picture.data:
            current_user.picture = save_picture(form.picture.data)

```

在 html 文件中调用 FileField。

Listing 7.28: Using FileField in html

```
<form method="POST" action="" enctype="multipart/form-data">
<!--这里必须给出编码方式enctype-->
  <fieldset>
    <div class="form-group">
        {{ form.picture.label() }}
        {{ form.picture(class="form-control-file") }}
        {% if form.picture.errors %}
            {% for error in form.picture.errors %}
                <span class="text-danger">{{ error }}</span></br>
            {% endfor %}
        {% endif %}
    </div>
  </fieldset>
</form>

```

7.3.11 Pagination

Pagination 可以减少 post 的内容，提高网站运行速度，同时避免页面过长，增加可读性。为了在分页时网页的内容不混乱，我们还需要对 post 的内容进行排序。

Listing 7.29: paginate

```
@app.route("/")
def home():
    page = request.args.get('page', default=1, type=int)
    posts = Post.query.order_by(Post.data)paginate(per_page=10, page=page)
    #以 10 个为 1 页，访问第 page 页
    return render_template('home.html', posts=posts)

```

在 html 文件中，可以设置一个导航栏，导航到指定页码，方便随机访问。

Listing 7.30: Using pagination in home.html

```
{% for page_num in posts.items %}
    <!--处理当前page的post内容-->
{% endfor %}

<!--导航栏-->
{% for page_num in posts.iter_pages(left_edfe=1, right_edge=1,
                                    left_current=1, right_current-2) %}
<!--这是一个预设，仅开头结尾和当前page前后的page_num会显示，其他page_num会缩成一个None-->
<!--right_current包含current page，故要比left_current多1-->
    {% if page_num %}
        {% if posts.page == page_num %} <!--把当前page的button设计的不一样-->
            <a
                class="btn btn-info mb-4"
                href="{{ url_for('home', page=page_num) }}">
                {{ page_num }}
            </a>
        {% else %}
            <a
                class="btn btn-outline-info mb-4"
                href="{{ url_for('home', page=page_num) }}">
                {{ page_num }}
            </a>
        {% endif %}
    {% else %}
        ... <!--缩成None的page_num显示成....-->
    {% endif %}
{% endfor %}

```

8 debug & tools

8.1 doctest

doctest 可以检查函数的输出，在代码注释样例中给出一组输入输出，若结果错误会报错。

Listing 8.1: doctest hello.py

```
from operator import floordiv, mod

def divide_exact(n, d):
  """Return the quotient and remainder of  dividing N by D
  >>> q, r = divide_exact(2013, 10)
  >>> q
  201
  >>> r
  2
  """
  return floordiv(n, d), mod(n, d)

```

Listing 8.2: doctest bash

```
python3 -m doctest hello.py
**********************************************************************
File "/mnt/d/Desktop/python/program file/test/test_basic/Pythonproject
1/hello.py", line 8, in hello.divide_exact
Failed example:
    r
Expected:
    2
Got:
    3
**********************************************************************
```

Table 8.1: Types of Error

错误名	错误类型	例子
Syntax Error	语法错误	代码结构不是 Python
IndentationError	缩进错误	缩进不一致或缺少缩进
TypeError	对象类型错误	将字符串与数字相加
NameError	命名错误	使用了尚未定义的变量或函数
Attribute	属性错误	使用了一个 class 没有的属性
IndexError	索引错误	访问超出 list 的索引位置
KeyError	键错误	访问字典中不存在的键
ValueError	值错误	将非数字字符串传给 int()
ImportError	导入模块错误	模块不存在或路径错误
ArithmeticError	数学错误	下面两个是他的子类
ZeroDivisionError	除以零	1/0
OverflowError	数值错误	算术结果超出数值类型范围
FileNotFoundError	文件不存在	以只读模式打开不存在的文件
IOError OSError	操作系统错误	文件操作出错
RuntimeError	标准错误以外的错误	‘raise’ 语句触发
AssertionError	断言错误	assert 语句触发
StopIteration	迭代器错误	迭代器没有更多项目供迭代

```
1 items had failures:
  1 of   3 in hello.divide_exact
***Test Failed*** 1 failures.
```

8.2 Try & Assert

try,except 用来处理可能出现报错的情况，并提供捕获错误的功能。
常见的错误类型如下，他们都有一个共同的父类 Exception

Listing 8.3: try except

```
try:
    f = open('testfile.txt'); val = bad_val
except FileNotFoundError as e: #捕获文件打开错误，并执行以下部分
    print(e) #输出 No such file or directory: 'testfile.txt'
except Exception: #捕获剩余的所有错误，exception 只会执行碰到的第一个
    print('Sorry. Something went wrong.')
else: #若 try 没有出现错误则执行
    print(f.read()); f.close()
finally: #无论是否出错都会执行
    print('Executing Finally...')
```

raise 语句可以在特定情况下手动报出错误

Listing 8.4: Raise an error

```
try:
    a = 2
    if a == 2:
        raise Exception
except Exception as e:
    print(e)
```

Python 中的 assert 语句类似于 C++assert 断言，不需要导入库，可以与 try，except 结合使用。

Listing 8.5: assert in python

```
def area_square(r):
    assert r > 0, 'A length must be positive'
    return r * r

try:
    area_square(-1)
except Exception as e:
    print(e)

```

8.3 Unit Testing

unittest 是标准库中的一个 module，可以用于代码分块 debug。unittest 中有很多 assert 类型，参考unittest 文档

Listing 8.6: Unit Testing

```
import unittest
import app #假设 app.py 是待测的文件

class TestApp(unittest.TestCase):

    @classmethod
    def setUpClass(cls): #setUpClass 会在所有 test 之前首先运行
        print('setupClass')

    @classmethod
    def tearDownClass(cls): #tearDownClass 会在所有 test 之后最后运行
        print('teardownClass')

    def setUp(self): #setUp 会在每个 test 之前运行一次，注意此函数名不能自定义
        self.eg1 = 10 #可以创建实例，初始化一些变量以简化 test 操作

    def tearDown(self): #tearDown 会在每个 test 之后运行一次，注意此函数名不能自定义
        pass #可以删除变量、实例以保证变量不影响下一个 test

    def test_add(self): #检测 add 函数，这里函数名必须以 test 开头
        result = app.add(self.eg1, 5) #检测 app.py 中的 add 函数
        self.assertEqual(result, 15)

    def test_div(self): #检测 div 函数
        sefl.assertRaises(ValueError, app.div, 10, 0)
        #检测 div(10, 0) 运行时是否会有 ValueError 报错
        with self.assertRaises(ValueError):
            app.div(10, 0) #这里利用文件管理器运行，与上一句效果相同

if __name__ == '__main__':
    unittest.main() #运行所有的 test

```

用以下语句运行测试

Listing 8.7: run unittest

```
python -m unittest test_app.py

```

对于网站申请类型的脚本，我们不希望 test 的通过与否取决于网站是否能够连接。为此，可以使用 mock module。

Listing 8.8: mock module

```
###main.py
import unittest
from unittest.mock import patch
from app import get_conn #假设需要检验 get_conn 函数
```

```
        class TestApp(unittest.TestCase):

            def test_get_conn(self):
                with patch('app.requests.get') as mocked_get: #若连接成功，会继续运行
                    mocked_get.return_value.ok = True
                    mocked_get.return_value.text = "Success!" #捏造一个成功连接，返回 Success
                    result = get_conn() #试运行 get_conn
                    mocked_get.assert_called_with('http://company.com/Schafer/May') #检验 url 的正确性
                    self.assertEqual(result, 'Success') #检验 get_conn 的返回值

        if __name__ == '__main__':
            unittest.main()

        ###app.py
        import requests

        def get_conn():
            response = requests.get('http://company.com/Schafer/May')
            if response.ok:
                return response.text
            else:
                return "Fail to connect!"

```

8.4 jupyter notebook

jupyter notebook 可以用于代码笔记，代码汇报等，提供代码实时运行的功能。可以在 ubuntu 中打开。

Listing 8.9: run jupyter notebok

```
        jupyter notebook

```

9 deep learning

9.1 pytorch 基本操作

9.1.1 数据操作

Listing 9.1: basic manipulation

```
          import torch

          x = torch.arange(12, dtype=torch.float32) #创建 1 至 12 的向量
          x = torch.zeros((3, 4)) #零张量
          x = torch.zeros_like(y) #创建同形状的零张量
          x = torch.ones((3, 4)) #全为 1 的张量
          x = torch.ones_like(y) #全为 1 的张量，形状同 y
          x_repeated = torch.repeat_interleave(input, repeats, dim=None)
              #input 为要进行重复的张量，repeats 为重复次数 (可以是整数或与 input 某维度长度相同的张量)
              #dim 指定在哪个维度上重复元素，若未指定，input 会被展平
          x_repeated = x.repeat(2, 1, 1) #在第 0 维重复 2 变，其它维不变
          x = torch.randn(3, 5) #均值为 0，标准差为 1 的高斯分布
          x = torch.tensor([[1, 3, 5, 6], [6, 9, 3, 5]]) #由 list 创建
          x = torch(2.0) #创建一个标量

          print(x.shape); print(x.numel()) #获取矩阵形状和总大小
          print(len(x)) #获取向量的维度或矩阵列向量的维度
          x = x.reshape(3, 4); x = x.reshape(3, 2, -1) #改变矩阵的形状，行优先
          x = x.unsqueeze(0) #在第 0 维插入一个新的维度
          x = x.permute((1, 0)) #将坐标轴顺序调换为 (1, 0)
          A = torch.cat(x, y, dim=0) #将两个列数相同的张量纵向拼接
```

```
        B = torch.cat(x, y, dim=1) #将两个行数相同的张量横向拼接
        A = torch.stack(x, y, dim=0) #将两个形状相同的矩阵堆叠，产生新维度 dim0

        print(x[2, 3]); x[2, 3] = 9 #通过行列索引访问和修改
        y = x[torch.tensor([1, 3, 5])] #通过一个 tensor 作为 index 取值
        y = x[[0, 1], [1, 0]] #取出 (0, 1) 和 (1, 0) 两个元素
        x[0: 2, :] = 12 #将前两行全部改成 12
        x = x[None, :] #在 axis=0 处加入一个维度
        x.fill_(10.0) #用 10.0 填充整个张量，可用于修改 0-dim tensor

        A = x.numpy(dtype=float); B = torch.tensor(A) #torch 和 numpy 转换
        A = x.detach().numpy() #detach 可以丢掉当前计算图，不参与梯度计算
        x.detach_() #原地操作丢掉当前计算图
        A = torch.tensor(data_frame.to_numpy()); B = pd.DataFrame(A.numpy()) #torch 和 DataFrame 转换
        val = a.item(); val = float(a); val = int(a) #大小为 1 的张量转标量
        x = x.type(y.dtype) #将 x 改为 y 的类型
        x_long = x.long() #修改数据类型为 long

        print(x + y, x - y, x * y, x / y, x ** y) #同形状对应元素运算，不同形状会广播
        print(x == y, x > y) #构建二元张量
        x[:] = exp(x) #求对应元素的 e 指数，用切片表示法可以节省内存

        x_sum = x.sum(); x_sum = x.sum(axis=[0, 1]) #求和，生成单元素张量
        x_sum_axis0 = A.sum(axis=0) #沿轴 0 降维，逐列求和，这里 axis 参数的轴会在降维后消失
        sum_x = x.sum(axis=1, keepdims=True) #沿轴 1 求和降维，但是轴数不变，可以广播
        x_mean_axis0 = A.mean(axis=0) #沿轴 0 降维，逐列求均值
        x_cumsum = torch.cumsum(x, dim=0) #计算张量在指定维度上的累积和，dim=0 即沿行累加
        x_max = x.max() #返回一个 tensor，最大值
        x_argmax = x.argmax() #返回一个 tensor，最大值对应的 index

        sorted_tensor, indices = torch.sort(x, dim=1, descending=True)
        #dim 默认为-1，indices 为排序后的张量中每个元素在原张量中 dim 维上的 index

```

注意，直接赋值不会重新分配内存

Listing 9.2: equal and clone

```
        a = torch.arange(10)
        b = a; print(a is b, a == b) #返回 True. True
        b = a.clone(); print(a is b, a == b) #返回 False, True

```

9.1.2 读写文件

Listing 9.3: File I/O

```
        import torch
        from torch import nn
        from torch.nn import functional as F

        x = torch.arange(4); y = torch.ones(3, 2)
        torch.save([x, y], 'x-file') #写入文件
        x2, y2 = torch.load('x-file') #读取文件

        mydict = {'x': x, 'y': y} #以字典形式读写
        torch.save(mydict, 'mydict')
        mydict2 = torch.load('mydict')

        net = Net() #nn.module 的子类
        torch.save(net.state_dict(), 'net.params')
        clone = Net(); clone.load_state_dict(torch.load('net.params'))
        clone.eval()

```

9.1.3 线性代数

Listing 9.4: linear algebra

```
A = A.T #矩阵转置
result = torch.dot(x, y) #求向量点积
y = torch.mv(A, x) #求矩阵-向量积
C = torch.mm(A, B) #求矩阵-矩阵积
C = torch.matmul(A, B) #矩阵乘法，可以处理多种输入形状，与基本相同
C = A @ B #矩阵乘法，在处理向量时会尝试将向量作为矩阵进行矩阵乘法，而 matmul 会返回内积
l2 = torch.norm(x) #求向量的 L_2 泛数
l1 = torch.abs(x).sum() #求向量的 L_1 泛数

```

9.1.4 微分

Listing 9.5: differentiation

```
x = torch.arange(4.0); x.requires_grad_(True) #注意这里必须是 float 向量而不是 int 向量
x = torch.arange(4.0, requires_grad=True) #两句等效，创建储存梯度的空间
y = 2 * torch.dot(x, x) #标量公式，不能是向量或张量，可以用 sum()
y.backward() #调用反向传播函数
print(x.grad) #输出梯度向量
x.grad.zero_() #在默认情况下，PyTorch 会累加梯度，我们需要清除之前的值
with torch.no_grad(): #在这个上下文中所有的操作不会计算梯度

y = x * x; u = y.detach() #求微分时将 y 是为一个常数，丢弃计算图中 y 的信息
z = u * x; z.sum().backward()
print(x.grad) #x.grad == u，全部为 True
x.grad.zero_()

```

9.1.5 概率

Listing 9.6: Probability and Statistics

```
import torch
from torch.distributions import multinomial
from d2l import torch as d2l

fair_probs = torch.ones([6]) / 6 #概率向量
sample_vec = multinomial.Multinomial(10, fair_probs).sample((500,))
    #进行 500 次测试，每次取 10 个样本，得到五行六列的矩阵

print(dir(torch.distributions)) #查看所有的函数和类

X = torch.normal(0, 1, (3, 5)) #生成正态分布数据，期望为 0，标准差为 1，3 行 5 列
X = torch.normal(0, 1, A.shape)

```

9.2 线性神经网络

9.2.1 线性回归

```
import torch
from d2l import torch as d2l
from torch import nn #neural network

true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000) #初始数据集

```

```
batch_size = 10
data_iter = d2l.load_array((features, labels), batch_size) #将初始数据转换成迭代器

net = nn.Sequential(nn.Linear(2, 1)) #权重矩阵的形状
net[0].weight.data.normal_(0, 0.01) #初始化权重，更多初始化方法参考torch.nn.init
nn.init.normal_(net[0].weight, mean=0, std=0.01) #这两句效果相同
net[0].bias.data.fill_(0) #初始化偏置
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
    #延迟初始化，只给定层的输出维数，在给定输入后根据输入的维数确定权重的 shape 并初始化

loss = nn.MSELoss() #定义损失函数为 L2 范数，更多损失函数参考torch.nn loss-functions
    #nn 中的损失函数由 reduction 选项，可选'mean', 'none', 'sum' 表示对损失值进行预操作，默认'mean'
    #注意 MSELoss 计算平方误差时不带系数 1/2
trainer = torch.optim.SGD(net.parameters(), lr=0.03) #设置优化算法为随机梯度下降

num_epochs = 3 #迭代次数
for epoch in range(num_epochs): #迭代训练
    for X, y in data_iter:
        l = loss(net(X), y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)

w = net[0].weight.data #获取权重和偏置数据数据
b = net[0].bias.data
print(net[0].weight.grad) #获取梯度数据
print(net[0].state_dict()) #获取一层的所有数据
iter_para = net.parameters() #获取一个迭代器，包含所有的参数 tensor
iter_name_para = net.name_parameters() #获取迭代一，所有的 (name, parameter) 对
print(net.eval()) #输出所有层的树，可视化
```

9.2.2 softmax 回归

由于计算精度的影响 softmax 函数会出现上溢或者下溢的情况，解决这个问题的技巧是:

- 在计算 softmax 之前，先从所有 o_k 中减去 $\max(o_k)$ 以解决上溢
- 将计算 softmax 和交叉熵结合在一起，保留 softmax 作为输出，并传递取对数的结果

$$\log(\hat{y}_j) = \log(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}) = o_j - \max(o_k) - \log(\sum_k \exp(o_k - \max(o_k)))$$

Pytorch 的 CrossEntropyLoss 可以接受网络的未归一化输出 **o**，并在其内部进行上述操作，输出 softmax 的结果和交叉熵损失。

Listing 9.7: Softmax

```
import torch
from torch import nn
from d2l import torch as d2l

train_iter, test_iter = d2l.load_data_fashion_mnist(256) #加载 Fashion-MNIST 图片集

net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10)) #创建权重变量

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights) #初始化权重

loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=0.1)
num_epochs = 10
```

```
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
d2l.plt.show()
```

9.2.3 多层向量机

Listing 9.8: activation function

```
import torch
from d2l import torch as d2l

x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x) #ReLU 函数，仅保留正值
y = torch.sigmoid(x) #sigmoid 压缩函数
y = torch.tanh(x) #双曲正切函数也能将实数域压缩到 (0, 1)
```

Listing 9.9: Multilayer

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(nn.Flatten(), #将矩阵展平成向量
                    nn.Linear(784, 256), #可选 bias=False，不带偏置
                    nn.ReLU(),
                    nn.Dropout(0.2), #标准暂退法正则化
                    nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);

batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
#parameters 表示模型中的所有计算梯度的参数
#这里可以设置 L2 范数惩罚
trainer = torch.optim.SGD(
    [{"params":net[0].weight, 'weight_decay':wd}, #给 weight 张量加上 L2 范数惩罚
     {"params":net[0].bias}], #net 是一个 OrderedDict，每个元素代表一层
    lr = lr
])

train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
d2l.plt.show()
```

9.3 获取数据集

9.3.1 线性回归数据集

Listing 9.10: Get linear data

```
def synthetic_data(w, b, num_examples):  #@save
    """生成y=Xw+b+噪声"""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

```
        def data_iter(batch_size, features, labels): #加载迭代器
            num_examples = len(features)
            indices = list(range(num_examples))
            # 这些样本是随机读取的，没有特定的顺序
            random.shuffle(indices)
            for i in range(0, num_examples, batch_size):
                batch_indices = torch.tensor(
                    indices[i: min(i + batch_size, num_examples)])
                yield features[batch_indices], labels[batch_indices]

```

9.3.2 Fashion-MNIST 图像分类数据集

Fashion-MNIST 有 10 个类别，每个类别 6000 张图片的训练数据集和 1000 张图片的测试数据集。

Listing 9.11: Download Fashion-MNIST

```
        import torch
        import torchvision
        from torch.utils import data
        from torchvision import transforms
        from d2l import torch as d2l

        d2l.use_svg_display()

        def get_DataLoader_mnist():
            return 4

        def load_data_fashion_mnist(batch_size, resize=None):
            trans = [transforms.ToTensor()] #将图像由 PIL 类型转换为 32 位浮点数，并除以 255 归一
            if resize: #调整图像大小以保证大小统一
                trans.insert(0, transforms.Resize(resize))
            trans = transforms.Compose(trans)
            mnist_train = torchvision.datasets.FashionMNIST(
                root='../data', train=True, transform=trans, download=True
            )
            mnist_test = torchvision.datasets.FashionMNIST(
                root='../data', train=False, transform=trans, download=True
            )
            return (
                data.DataLoader(mnist_train, batch_size, shuffle=True,
                                num_workers=get_dataloader_worker()),
                data.DataLoader(mnist_train, batch_size, shuffle=False,
                                num_workers=get_dataloader_worker())
            ) #采用 4 个进程来读取数据

```

9.3.3 数据加载切分

- torch.utils.data 可以用于切分数据。
- torch.utils.data.DataLoader 接收数据集 (需要有 __len__ 和 __getitem__)，并返回一个 iterator
- torch.utils.data.Dataset 是一个基类,数据集类可以是他的子类(当然也可以是 list,torch.tensor 等)

Listing 9.12: DataLoader

```
        import torch
        import torch.utils.data

        # 自定义数据集
        class CustomDataset(torch.utils.data.Dataset):
```

```
            def __init__(self, data, labels):
                self.data = data
                self.labels = labels

            def __len__(self):
                return len(self.data)

            def __getitem__(self, index):
                # 返回指定索引的数据和标签
                return self.data[index], self.labels[index]

        # 创建数据和标签
        data = torch.randn(100, 3)  # 100 个样本，每个样本 3 个特征
        labels = torch.randint(0, 2, (100,))  # 100 个标签

        # 创建自定义数据集对象
        dataset = CustomDataset(data, labels)

        # 使用 DataLoader 加载自定义数据集
        dataloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True)

```