Python 学习笔记

Yusong

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
>>> 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 condition statements

Listing 1.3: 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

1.3 iteration

Listing 1.4: iteration

```
i, total = 0, 0
while i < 3:
  total += i
  i += 1

for i in range(3):
  total += i
```

1.4 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。

1.4.1 定义函数

Listing 1.5: 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
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

```

1.4.2 调用函数

以下是几种正确的调用方法

Listing 1.6: 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 1.7: 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)

```

1.4.3 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 1.8: 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 1.9: 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
```

1.4.4 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 1.10: static functions

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

1.4.5 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 1.11: 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)

```

1.4.6 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 1.12: 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 1.13: 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)

```

1.4.7 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 1.14: self-reference

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

```

1.4.8 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 1.15: 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')

```

1.4.9 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 1.16: 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))

```

1.4.10 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 1.17: 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

```

1.5 input and output

1.5.1 打开关闭文件

Listing 1.18: 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 1.1: 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	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	

1.5.2 读取写入文件

Listing 1.19: 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)

1.5.3 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 1.20: contextmanager

```
import contextlib

@contextlib.contextmanager
def open_file(file, mode):
    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!')

```

1.5.4 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 1.21: 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})

```

1.5.5 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 1.22: 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

```

2 Container

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。

Listing 2.1: iterator

```
ite = map(lambda x: x * x, range(10), [i for i in range(20)])
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
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。

```

2.2 list

Listing 2.2: 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 2.3: 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 2.4: 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 2.5: 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
david_house=student1.get("David", "Not Found") #访问字典，找不到返回 Not Found
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
student1["Hermione"]="Gryffindor" #加入元素
student1.update({"Hermione":"Gryffindor"}) #合并两字典
for key,value in student1.items() #遍历字典
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 2.6: 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 2.7: 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 2.8: 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 2.9: 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 2.10: 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)

```

3 regular expression

3.1 grammar

正则表达式由元字符，限定符，选择字符，排除字符等组成。

3.1.1 元字符

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 3.1: 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

3.1.2 限定符

Table 3.2: 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

3.1.3 brackets

- 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。

3.1.4 notice

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’。

3.2 re module

Python 中的 re 模块可以通过正则表达式处理字符串。

3.2.1 匹配模式

Table 3.3: 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

3.2.2 匹配函数

这里匹配的字符串区间没有重叠的部分。

Listing 3.1: string matching

```python
import re #导入模块

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')]

```

3.2.3 替换字符串

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 3.3

Listing 3.2: string substitution

```python
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********'

```

3.2.4 切分字符串

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 3.3

Listing 3.3: 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 Class

4.1 定义类模板

4.1.1 基本结构

Listing 4.1: 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.")

```

4.1.2 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。

4.1.3 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 4.2: 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 4.3: 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 4.4: deleter

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

4.1.4 类的特殊成员 (魔术方法)

Listing 4.5: 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})")
        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

```

4.2 类的使用

4.2.1 创建类的实例

Listing 4.6: Create an instance of class

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

```

4.2.2 类成员的使用

Listing 4.7: 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

```

4.3 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 4.8: 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。

4.4 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 4.9: 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

```

4.5 类的典型例子

4.5.1 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 4.10: File management class

```
class Open_File():

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

    def __enter__(self):
        self.file = open(self.filepath, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, traceback)
        self.file.close()

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

```

5 basic module

5.1 install and import module

5.1.1 install and manage modules

可以通过 pip 安装和管理所需的 module。

Listing 5.1: 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

```

5.1.2 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 5.2: 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 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

```

5.1.3 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。

Listing 5.3: 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 5.4: activate.d/env_vars.sh

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

```

Listing 5.5: 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 5.6: 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

```

5.1.4 import modules

Listing 5.7: import module

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

```

5.1.5 view the functions in modules

Listing 5.8: view the functions in builtin module

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

```

5.2 builtins

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。

5.3 sys

有关系统操作的 module

Listing 5.9: 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

```

5.4 os

os module 可以实现操作系统相关的功能。

Listing 5.10: 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
        #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
        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')

```

5.5 logging

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 5.11: logging module

```
        logging.basicConfig(filename='app.log', filemode='w', level=logging.DEBUG,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S'
                    )
        #以下信息将会以上面设定的形式存到 app.log 文件中，只有 DEBUG 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")
```

```
#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
    ]
)

```

5.6 date

5.6.1 datetime module

datetime 用于时间和日期戳记录和转换

Listing 5.12: 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.isoformat()) #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

```

5.6.2 pytz module

pytz 用于管理时区 (time zone) 相关的信息。

Listing 5.13: 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

```

5.6.3 calendar

calendar 用于计算天数，星期数，星期几等数据。

Listing 5.14: 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

```

5.7 calendar

5.8 time

time module 可以用于计时

Listing 5.15: time module

```
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

```

5.9 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 5.16: 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
collection.Counter(['a', 'b', 'c', 'a', 'b', 'b'])

#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

```

5.10 random

Listing 5.17: 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

```

5.11 cn2an

cn2an module 可以实现中文数字到阿拉伯数字的转换。

Listing 5.18: cn2an

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

```

6 Maths and Statistics

6.1 math

math module 提供了最基础的数学函数。

Listing 6.1: 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.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

```

6.2 numpy

6.2.1 定义矩阵

Listing 6.2: 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.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

```

6.2.2 特殊函数

Listing 6.3: 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
(n,m)=np.where(X>1) #查找满足条件的元素坐标
a=X[n,m] #取出满足条件的元素
l=np.argwhere(X>1) #n 为坐标组成的二维 array
```

6.2.3 矩阵运算

Listing 6.4: calculation of matrix

```
c=np.dot(x,y) #矩阵乘法
c=x*y #对应元素相乘
c=np.dot(a,np.linalg.inv(b)) #矩阵右除
c=np.dot(a,np.linalg.inv(a),(b)) #矩阵左除
c=np.transpose(a);c=a.T #矩阵转置
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
matrix.sum(axis=0) #求和, 1 行 0 列
```

6.3 Pandas

6.3.1 Series 类型

Series 类型是一维数组，由 index 和 value 组成。

Listing 6.5: Series in Pandas

```
import pandas

data=['A','B','C']
index=['a','b','c']

series = pandas.Series(data) #默认 index 从 0 开始编号
series_with_index = pandas.Series(data,index=index) #规定 index
print(series.index,series.value) #会输出数组
print(series_with_index['a']) #调用 Series 的元素
```

6.3.2 DataFrame 类型

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 6.6: Create a dataframe

```
import pandas

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 = pandas.DataFrame(data) #创建 DataFrame 对象, 默认 index 从 0 开始编号
data_frame = pandas.DataFrame(data,index=index) #规定 index
data_frame = pandas.DataFrame(data,columns=['a','b']) #指定列
print(data_frame)
```

6.3.3 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 6.7: Read and write files using Pandas

```
import pandas

data = pandas.read_csv(<filename>)
data = pandas.read_excel(<filename>)
data.to_csv(<new_filename>,columns=['A','B'],index=False) #不写入行索引
data.to_excel(<new_filename>,columns=['A','B'],index=False) #写入 excel 文件

```

6.3.4 基本操作

Listing 6.8: Basic functions about DataFrame

```
data_frame['d']=[50,60,70,80,90] #增添数据

data_frame.drop([0,1],inplace=True) #按 index 删除，inplace 表示对原数据删除不返回删除后的对象
data_frame.drop(labels='a',axis=1,inplace=True) #按 column 删除，axis=1 表示列，0 表示行

data_frame['a'][1] = numpy.nan; data_frame['b']=[7,8,9,10,11] #修改数据
data_frame.a[1] = numpy.nan; data_frame.b = [7,8,9,10,11] #这与上面是等价的

```

6.3.5 统计操作

Listing 6.9: Perform statistical operations using DataFrame

```
#数据预处理
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,inplace=True) #删除包含空缺值的整行数据
data_frame.fillna(0,inplace=True) #修改空缺值
data_frame.fillna({'A':0,'B':1,'C':2},inplace=True) #每列空缺值用指定的值代替

#常用的统计函数
average = data_frame.mean() #求每列的平均值，输出一个 Series
score = data_frame.a+data_frame.b-data_frame.c #可以直接做向量运算
data_frame.sort_values(['a'],axis=0,ascending=False,inplace=True) #排序，ascending 为升序

```

7 tools

7.1 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 7.1: 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
```

Table 7.1: 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

```
    """
    return floordiv(n, d), mod(n, d)

```

Listing 7.2: 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
**********************************************************************
1 items had failures:
  1 of   3 in hello.divide_exact
***Test Failed*** 1 failures.

```

7.2 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 7.3: 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 7.4: 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 7.5: 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)

```

7.3 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 7.6: run jupyter notebok

```
        jupyter notebook

```

8 visualizations

8.1 matplotlib

8.1.1 画布预处理

Listing 8.1: basic setup of plt

```
          from matplotlib import pyplot as plt
          import numpy as np

          plt.figure(figsize=(10,20),facecolor,edgecolor)
          plt.title("title")
          plt.xlabel("x");plt.ylabel("y")
          plt.style.use("seaborn-v0_8")
          plt.legend() #显示图例
          plt.xticks(ticks=[2*i+1 for i in range(10)],labels=[2*i+1 for i in range(10)])
          plt.xlim(2,22)
          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,color,linestyle,linewidth) #水平参考线
```

```
            plt.axvline(10,color,linestyle,linewidth) #垂直参考线
            plt.axhspan(5,7,color,linestyle,linewidth) #水平参考区域
            plt.axvspan(10,12,color,linestyle,linewidth) #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

```

8.1.2 基本图表

Listing 8.2: plots of plt

```
            plt.plot(x,y,color="red",
                    linestyle="dashed", #or dotted or dashdot
                    linewidth=3,
                    marker=".", #or "," "o" "+" "x"
                    markersize=8,
                    markerfacecolor="blue",
                    markeredgecolor="cyan"
            )
            plt.bar(x,y,width,bottom=3, #柱形底部高度
                    hatch="/") #or "l" ""\\" "//"
            plt.barh(x,y)
            plt.hist(x,bins) #直方图，bins 可以是整数（条数）或列表
            plt.scatter(x,y,s) #s is a list, which stands for the size of the dots
            plt.pie(x,colors=["red","blue","yellow"],
                    autopct=%1.1f%%, #整数部分一位，小数部分一位
                    explode=[0,0.5,0], #将第二块拉出 0.5
                    shadow=True,labels=["一月","二月","三月"],
            ) #饼图
            plt.pie(x,colors=["red","blue","yellow"],
                    autopct=%1.1f%%, #整数部分一位，小数部分一位
                    explode=[0,0.5,0], #将第二块拉出 0.5
                    shadow=True,labels=["一月","二月","三月"],
                    radius=1.0,wedgeprops={"width":0.6} #内外圆半径
            ) #圆环图
            plt.boxplot(x,showmeans=True, #显示均值
                        flierprops={"marker":"o", #or "," "+" "x"
                                    "markerfacecolor":"red",
                                    "markeredgecolor":"black",
                                    "markersize":8
                        }, #异常点样式
                        patch_artist=True, #自定义箱型
                        boxprops={"facecolor":"red",
                                  "edgecolor":"yellow"
                        } #箱型样式
            ) #箱型图
            plt.stackplot(x,y1,y2,y3,color=["red","yellow","blue"])
            #面积图（可堆叠）
            plt.errorbar(x,y,yerr=[lower_errors,upper_errors],
                        ecolor=blue, #color of the errorbars
                        elinewidth=3, #width of the errorbars
                        capsize=2 #横杠大小
            )
            plt.imshow(x, #x 是个二维列表
                       cmap=plt.cm.cool #设置颜色
            ) #绘制热力图
            plt.colorbar() #显示图例

```

8.1.3 极坐标图表

Listing 8.3: 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'])

```

8.1.4 三维图表

Listing 8.4: 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

```

8.2 wordcloud

Listing 8.5: 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()

```

9 web spider

9.1 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。

9.1.1 urllib module

利用 request 模块可以实现 get 请求方式获取网页内容。

Listing 9.1: 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 9.2: 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)
```

9.1.2 urllib3 module

urllib3 是一个更强大的 Python 库。

Listing 9.3: 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
```

9.1.3 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 9.4: 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
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）
```

为了绕开反爬设计，我们可以请求 header。

Listing 9.5: 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)

```

9.2 处理 html 文件

BeautifulSoup module 用于处理 html 文件

Listing 9.6: 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

```