



廣東白雲學院
GUANGDONG BAIYUN UNIVERSITY

学习总结

题 目：《Python 学习总结》

姓 名：NYC

专业班级：B23 计算机科学与技术 1 班

学 院：大数据与计算机学院

指导教师：李妍

2024 年 5 月

目录

摘要	6
第 1 章 初识 Python 的世界.....	7
1.1 Python 语言特点及其应用	7
1.2 Python 的注释	7
1.2.1 单行注释.....	7
1.2.2 多行注释.....	8
第 2 章 Python 基础知识.....	9
2.1 基本语法.....	9
2.2 数据类型.....	10
2.3 标识符.....	10
2.4 运算符.....	10
2.4.1 算术运算.....	10
2.4.2 逻辑运算.....	11
2.5 运算符的优先级.....	11
第 3 章 Python 字符串输入和输出.....	13
3.1 转义字符.....	13
3.2 字符串格式输出.....	14
3.3 字符串运算.....	14
第 4 章 Python 的组合数据类型.....	16
4.1 概述.....	16
4.2 列表（list）	16
4.2.1 创建列表 list.....	16
4.2.2 追加与删除 list 的元素.....	17
4.3 元组（tuple）	19
4.3.1 创建元组 tuple.....	19
4.4 字典（dict）	20
4.4.1 创建字典.....	20
4.4.2 dict.get()函数	21

4.4.3 dict.pop()函数.....	21
4.5 集合 (set)	22
4.5.1 创建集合.....	22
4.5.2 添加与删除集合元素.....	23
第 5 章 Python 程序的流程控制.....	24
5.1 条件判断.....	24
5.1.1 双选择结构.....	24
5.1.2 多选择条件结构.....	24
5.2 循环结构.....	25
5.2.1 for 循环语句.....	25
5.2.2 while 循环语句.....	26
第 6 章 用函数实现代码复用	27
6.1 函数的定义和调用.....	27
6.2 函数的参数.....	27
6.2.1 默认参数.....	27
6.2.2 可变与不定长参数.....	29
6.2.3 关键字参数.....	31
6.3 递归函数.....	32
6.4 匿名函数.....	32
第 7 章 正则表达式	33
7.1 正则表达式的语法和规则.....	34
7.1.1 普通字符.....	34
7.1.2 特殊字符.....	34
7.1.3 限定符.....	34
7.1.4 定位符和其他.....	35
7.2 正则表达式的应用.....	35
7.2.1 re 模块	35
7.2.2 match()函数	35
7.2.3 search()函数.....	36

7.2.4 split()函数	36
7.2.5 sub()函数	37
第 8 章 面向对象编程	38
8.1 类和实例.....	38
8.1.1 创建类和属性.....	38
8.1.2 实例属性.....	38
8.1.3 数据封装(类方法).....	40
8.1.4 访问限制.....	41
8.2 继承与多态.....	44
8.2.1 继承.....	44
8.2.2 多态.....	45
第 9 章 文件操作	46
9.1 文件的读写.....	46
9.1.1 读文件.....	46
9.1.2 readline()方法	48
9.1.3 关闭文件.....	49
9.1.4 写文件.....	50
9.2 操作文件和目录.....	50
9.2.1 删除与重命名文件.....	51
9.2.2 创建和删除文件夹.....	51
第 10 章 Python 异常处理.....	52
10.1 概念.....	52
10.2 错误（异常）处理.....	52
10.2.1 python 标准异常	53
10.2.2 抛出异常（错误）	57
第 11 章 Python 的模块使用与程序打包.....	58
11.1 模块的概念.....	58
11.2 模块导入.....	58
11.2.1 import 语句	58

11.2.2 from import 语句	59
11.3 创建自定义模块.....	60
11.4 程序打包.....	62
11.4.1 pyinstaller 的使用	62
结语	63
致谢	64
参考文献	65

摘要

本文章旨在深入探讨 Python 编程语言的核心概念及其在实际应用中的高效性。通过系统学习 Python 的基础知识、字符串处理、组合数据类型、流程控制、函数封装、正则表达式、面向对象编程等关键特性，本文详细阐述了 Python 编程的方法和技巧。研究采用案例分析法，结合实际编程任务，对 Python 的文件操作、异常处理以及模块使用等高级特性进行了深入剖析。结果表明，Python 作为一种简洁、易学且功能强大的编程语言，在提升编程效率、简化代码逻辑以及应对复杂问题方面具有显著优势。本文为 Python 编程的初学者和研究人员提供了一个全面的学习框架，也为进一步探索 Python 在数据科学、人工智能等领域的应用奠定了基础。

关键词：Python 编程；字符串处理；面向对象；异常处理；模块使用

第1章 初识 Python 的世界

1.1 Python 语言特点及其应用

1. 易学：简洁的语法和少量关键字，易于上手。
2. 开源：免费使用，可自由修改和分发源代码。
3. 维护性：代码清晰，便于维护。
4. 丰富的库：标准库广泛，跨平台兼容性好。
5. 可移植：适用于多种平台。
6. 互动性：支持交互式编程，便于测试和调试。
7. 可扩展：可通过 C/C++ 扩展以提升性能或保护算法。
8. 面向对象：结合了过程式和面向对象编程，实现简单而强大。
9. GUI 支持：方便创建和移植图形用户界面。
10. 嵌入性：可嵌入 C/C++ 程序，提供脚本化能力。

1.2 Python 的注释

Python 注释是用来标记和解释程序解释器会自动忽略掉注释掉的语句，这会增加代码整体的可读性。

1.2.1 单行注释

开头以“#”字后面接语句，不得窜行。例图 1-1：

```
# 计算n!使用递归函数调用
def fact(n):
    if n==1 or n==0:
        return 1
    return n*fact(n-1)
print(fact(3)) # 函数调用通过栈数据类型实现的
```

图1-1

1.2.2多行注释

多行注释用三个单引号或者三个双引号将注释括起来。例如图 1-2:



The diagram illustrates multi-line comment syntax in a code editor. On the left, line numbers 4 through 8 are listed. A vertical line separates the line numbers from the code content. At line 4, there is a dropdown arrow and three single quotes. At line 5, there is a space. At line 6, the text '多行注释' is written. At line 7, there is a space. At line 8, there are three single quotes.

```
4  ✓  '''  
5  
6  多行注释  
7  
8  '''
```

图1-2

第2章 Python 基础知识

2.1 基本语法

1. Python 语句的缩进：缩进的语句视为代码块，每段代码块的空白数量可以是任意的，但要确保同段的代码块语句需要保持相同的空白数量。按照约定成熟的惯例，应该使用一个 Tab（制表符）也就是四个空格的缩进。例图 2-1。

```
6      a=int(input("请输入a值:"))
7      if a>=0:#判断A>=0就输出a 否则输出-a
8          print(a)
9      else:
10         print(-a)
```

图2-1

2. 下划线 “_” 的等价：就如下面的例图 2-2，“a=1000000 b=1_000_000” 这两者是相等的。运行结果如图 2-3 所示。

```
10      a=1000000 #a=1000000 b=1_000_000这两者是相等的—
11      b=1_000_000
12      print(a==b)
13      print(f'a={a}\nb={b}')
```

图2-2

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
True
a=1000000
b=1000000
```

图2-3

2.2数据类型

- 1. 整数类型：指定为“int”类型，包含正整数和负整数，默认用十进制来表示。
- 2. 浮点型：浮点数就是带有小数的数字类型，例如：“3.1415926”、“-9.231”。
- 3. 字符串：以单引号或者双引号括起来的任意文本，例如：“abc”，注意：用双引号或单引号括起来的数字是没有大小数值关系的，即只表示为文本。
- 4. 布尔值：只有“True”也会被当成整型中的 1，“False”会被当成 0。布尔值还可以用“and”与运算、“or”或运算和“not”非运算来运算在。
- 5. 空值：Python 中特殊的值，用“None”来表示。不能与 0 相提并论，0 是有数值意义的，而“None”是特殊的。

2.3标识符

标识符名必须是大小写字母、数字和下划线“_”的组合，且不能用数字开头。例图 2-4。

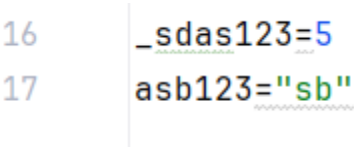


图2-4

2.4运算符

2.4.1算术运算

运算符	说明
+	相加
-	相减
*	相乘
/	相处
//	整除
%	求余

**	求幂次方
----	------

例图 2-5：表示 2 的 5 次方结果为 32。

```

12
13     a=2 ** 5
14     print(a)

```

图2-5

2.4.2逻辑运算

包含“and”、“or”、“not”，分别为与、或、非运算，结果为布尔值“True”、“False”。

例图 2-6，运行结果如图 2-7。

```

13     a=1
14     b=2
15     c=0
16     print(a or b,a and c,not a)

```

图2-6

```

D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
1 0 False

进程已结束，退出代码为 0

```

图2-7

2.5运算符的优先级

优先级从高到低排序

运算符	描述
**	指数（最高优先级）
~ + -	按位翻转，一元加号和减号（最后两

	个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
not	逻辑非
and	逻辑与
or	逻辑或

第3章 Python 字符串输入和输出

3.1转义字符

若需要输入一些特殊字符，则需要在特殊字符前加入反斜杠“\”即转移字符。

常用的转义字符

转义字符	描述
\\(在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数, y 代表 0~7 的字符, 例如: \012 代表换行。
\xyy	十六进制数, 以 \x 开头, yy 代表的字 符, 例如: \x0a 代表换行
\other	其它的字符以普通格式输出

3.2字符串格式输出

1. Format()法：传入的参数依次替换字符串的占位符{0}、{1}……。例图 4-1，结果如图 4-2。

```
11 # format()法
12 print('Hello {0},you got {1:.1f}'.format(*args: 'sb', 12.87))
13
```

图4-1

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
Hello sb,you got 12.9
```

图4-2

2. f-string 法：以 f 开头的字符串，若字符串包含{x}，就会以对应的变量替换。例图 4-3，结果如图 4-4。

```
16 # f-string
17 a = "Hello World"
18 b = 9.9
19 print(f"Hi {a} and your grade is {b:.2f}%")
20
```

图4-3

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
Hi Hello World and your grade is 9.90%
```

图4-4

3.3字符串运算

常用的字符串的控制符

操作符	描述
+	字符串连接
*	重复输出字符串

[]	索引字符串中的字符
[:]	截取字符串中的一部分，遵循左闭右开原则
in	字符串中包含给定的字符返回 True
not in	字符串中不包含给定字符串返回 True
R/r	转义所有字符串

例图 4-5，结果为图 4-6。

```

22     a = "Hello"
23     b = "Python"
24
25     print("a + b 输出结果: ", a + b)
26     print("a * 2 输出结果: ", a * 2)
27     print("a[1] 输出结果: ", a[1])
28     print("a[1:4] 输出结果: ", a[1:4])
29
30     if ("H" in a):
31         print("H 在变量 a 中")
32     else:
33         print("H 不在变量 a 中")
34
35     if ("M" not in a):
36         print("M 不在变量 a 中")
37     else:
38         print("M 在变量 a 中")
39
40     print(r'\n')
41     print(R'\n')

```

图4-5

```

D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
a + b 输出结果:  HelloPython
a * 2 输出结果:  HelloHello
a[1] 输出结果:  e
a[1:4] 输出结果:  ell
H 在变量 a 中
M 不在变量 a 中
\n
\n

```

图4-6

第4章 Python 的组合数据类型

4.1 概述

1. 组合数据类型可分为：序列类型、映射类型和集合类型。
2. 序列：包含字符串、列表和元组 3 组。
3. 映射类型：用键值映射数据，最具代表性的映射类型是字典。
4. 集合类型：元素是无序的，集合内不允许有相同元素存在。

4.2 列表（list）

4.2.1 创建列表 list

用“[]”创建列表。例图 4-1。1 其中 list 中包含 4 个元素“sb0”为第 0 个元素，依此类推。

```
classmate = ['sb0', 'sb1', 'sb2', 'sb3']
print(classmate)

# 总共有几个元素
print(len(classmate))

# 第一个和最后一个元素
print(classmate[0], classmate[3])

# 最后一个元素是
print(classmate[-1])
```

结果：

```
D:\Python3.12.2\python.exe "D:\NAS\My c
['sb0', 'sb1', 'sb2', 'sb3']
4
sb0 sb3
sb3
```

图4-1

4.2.2追加与删除 list 的元素

list 是一个可变的有序表，可以往 list 中追加元素到末尾用在 list 名字后面加个.append(“要插入的元素”)。例图 4-2。

```
classmate = ['sb0', 'sb1', 'sb2', 'sb3']  
# 追加list的元素到classmate  
classmate.append('jenny')  
print(classmate)  
print(len(classmate))
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"  
['sb0', 'sb1', 'sb2', 'sb3', 'jenny']  
5
```

图4-2

同时我们也可以把元素插入到指定位置，例将“JACK”插入到列表索引 1 中，如图 4-3。

```
# 把元素插入到指定位置  
classmate.insert(__index: 1, __object: 'JACK')  
print(classmate)
```

结果如下：

```
['sb0', 'JACK', 'sb1', 'sb2', 'sb3', 'jenny']
```

图4-3

要删除 list 末尾的元素，得在 list 名后加.pop()来删除，若在 pop()上给定参数则可以删除 list 中指定位置元素，同时想要删除指定元素就得用 remove(“指定元素”)。例图 4-4。

```
31 # 删除list末尾元素
32 classmate.pop()
33 print(classmate)
34
35 # 删除list指定位置元素pop[i]
36 classmate.pop(1)
37 print(classmate)
38
39 # 删除指定元素
40 classmate.remove('sb0')
41 print(classmate)
```

结果如下：

```
['sb0', 'JACK', 'sb1', 'sb2', 'sb3', 'jenny']
['sb0', 'JACK', 'sb1', 'sb2', 'sb3']
['sb0', 'sb1', 'sb2', 'sb3']
['sb1', 'sb2', 'sb3']
```

图4-4

如果想直接替换 list 中的元素成其他元素可以用赋值的方式替换。例图 4-5。

```
# 直接对list的元素替换成别的元素
classmate[1] = 'Babe'
print(classmate)
```

结果如下：

```
['sb1', 'Babe', 'sb3']
```

图4-5

4.3元组 (tuple)

tuple 与 list 相似但 tuple 初始化之后就不能修改了也就没有了.append .insert 等修改增加 list 的方法，书写上把 list: []变成了 tuple: ()

4.3.1创建元组 tuple

要创建元组 tuple 书写上把 list: []变成了 tuple: (), 例图 4-6。

```
# tuple 与list相似但tuple初始化之后就不能修改了也就没有.append .insert  
classmate_tuple = ('李鸿章', '曾国藩', '林则徐') # 初始化元组  
# 因为tuple不可变, 所以代码更安全。如果可能, 能用tuple代替list  
print(classmate_tuple)
```

结果如下:

```
D:\Python3.12.2\python.exe "D:\N  
( '李鸿章', '曾国藩', '林则徐' )
```

图4-6

虽然元组不能修改但是可以嵌套可变 list, 例图 4-7。

```
# tuple 里面是可以嵌套可变list的  
classmate_tuple_list = ('sb0', 'sb1', ['sb3', 'sb4'])  
# tuple包含了3个元素分别是sb0,sb1,list  
print(classmate_tuple_list)  
classmate_tuple_list[2][0] = 'sb2'  
classmate_tuple_list[2][1] = 'sb3'  
# 变的不是tuple而是tuple变化的是里面的list  
print(classmate_tuple_list)
```

结果如下:

```
( 'sb0', 'sb1', [ 'sb3', 'sb4' ] )  
( 'sb0', 'sb1', [ 'sb2', 'sb3' ] )
```

图4-7

4.4字典（dict）

使用键-值（key-value）组成来存储，可以通过 key 找到映射的 value，具有极快的查找速度。

4.4.1创建字典

字典中的每个元素都包含键和值两部分，创建字典的规则：变量: dict[类型A,类型 B]={‘A’: B}。例图 4-8。

```
# 变量: dict[类型A,类型B]={‘A’: B}
d1: dict[str, int] = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
print(d1)
print(d1['Michael'])
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.p
{'Michael': 95, 'Bob': 75, 'Tracy': 85}
95
```

图4-8

除了 dict 的方法，除了初始化时指定外，还可以通过 key 放入。一个 key 只能对应一个 Value 如果重复给值则会被替换。例图 4-9。

```
# dict的方法，除了初始化时指定外，还可以通过key放入
d1['sb'] = 250
print(d1)
print(f"sb={d1['sb']}")
# 一个key只能对应一个Value 如果重复给值则会被替换
d1['sb'] = 88
print(d1)
print(f"sb={d1['sb']}")
```

结果如下：

```
{'Michael': 95, 'Bob': 75, 'Tracy': 85, 'sb': 250}  
sb=250  
{'Michael': 95, 'Bob': 75, 'Tracy': 85, 'sb': 88}  
sb=88
```

图4-9

4.4.2dict.get()函数

可以通过 dict 提供的 get()方法, 如果 key 不存在, 可以返回 None, 或者自己指定的 value

get()方法如果 key 不存在, 返回 None (在交互式不显示)。例图 4-10。

```
# get()方法如果key不存在, 返回None (在交互式不显示)  
print(d1.get('nobody'))
```

结果如下:

```
# get()方法如果key不存在, 返回None (在交互式不显示)  
print(d1)  
print(d1.get('nobody'))
```

图4-10

4.4.3dict.pop()函数

若要删除一个 Key 用 pop()对应的 value 也会从 dict 中删除。例图 4-11。

```
# 若要删除一个Key用pop()对应的value也会从dict中删除  
print("删除前", d1)  
d1.pop('sb')  
print("删除后", d1)
```

结果如下:

```
# 若要删除一个Key用pop()对应的value也会从dict中删除  
print("删除前", d1)  
d1.pop('sb')  
print("删除后", d1)
```

图4-11

4.5集合（set）

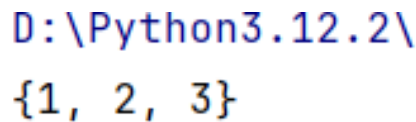
set 和 dict 类似，也是一组 key 的集合，但不存储 value。由于 key 不能重复，所以，在 set 中，没有重复的 key。

4.5.1创建集合

要创建一个 set，需要提供一个 list 作为输入集合，例图 4-12 中只是告诉你这个 set 内部有 3 个元素显示的顺序也不表示 set 是有序的。

```
22  # set key的集合，但不存储value
23  # 由于key不能重复，所以，在set中，没有重复的key
24  s = set([1, 2, 3])
25  print(s)  # 只是告诉你这个set内部有3个元素显示的顺序也不表示set是有序的
```

结果如下：



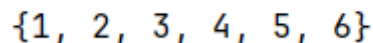
```
D:\Python3.12.2\
{1, 2, 3}
```

图4-12

图 4-13 中重复的元素会被过滤掉，彰显出集合的特点。

```
# 重复的元素会被过滤掉
s = set([1, 1, 2, 2, 3, 4, 5, 5, 6])
print(s)
```

结果如下：



```
{1, 2, 3, 4, 5, 6}
```

图4-13

4.5.2 添加与删除集合元素

通过 `add(key)` 方法可以添加元素到 `set` 中，例图 4-14。

```
print(s)
# .add(key) 来添加元素到set当中
s.add(8)
print(s)
```

结果如下：

```
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6, 8}
```

图4-14

通过 `remove(key)` 方法可以删除元素，例图 4-15。

```
print(s)
# .remove(key) 删除set中的元素
s.remove(4)
print(s)
```

结果如下：

```
{1, 2, 3, 4, 5, 6, 8}
{1, 2, 3, 5, 6, 8}
```

图4-15

第5章 Python 程序的流程控制

5.1 条件判断

5.1.1 双选择结构

Python 通过 if 语句通过判断条件表达式来判断真假。根据 Python 的缩进规则，如果 if 语句判断是 True 则执行语句否则执行 else 语句。例图 5-1。

```
23     age = 3
24     if age >= 18:
25         print('your age is', age)
26         print('adult')
27     else:
28         print('your age is', age)
29         print('teenager')
```

结果如下：

```
your age is 3
teenager
```

图5-1

5.1.2 多选择条件结构

使用 if……elif……elif 等多个 elif 来判断选择多个分支，例图 5-2 其中若输入值为大于等于执行第一个语句，若大于等于 6 小于 8 则执行第二个语句最后这些条件都不满足则执行 else 语句。

```
# 从上往下判断如果其中一个判断为True则不会执行下面的elif判断
age = int(input('Enter your age: '))
if age >= 18:
    print("Your age is %d,you are adult" % age) # 一般格式输出
elif 6 <= age < 18:
    print(f"Your age is {age},you are teenager") # f-string用法
else:
    print("Your age is {0},you are child".format(age)) # format用法
```


结果如下：

```
Enter your age: 19
Your age is 19,you are adult
Enter your age: 7
Your age is 7,you are teenager

Enter your age: 3
Your age is 3,you are child
```

图5-2

5.2循环结构

5.2.1for 循环语句

Python 的循环有两种，一种是 for...in 循环，依次把 list 或 tuple 中的每个元素迭代出来，例图 5-3。

```
names = ['jack', 'lucy', 'marry']
for name in names: # 依次输出names的元素
    print(name)
```

结果如下：

```
D:\Python3.12.2\p
jack
lucy
marry
```

图5-3

第二种则是循环就是把每个元素代入某一个变量，然后执行缩进块的语句。
例图 5-4 中 1-10 的整数之和，可以用一个 sum 变量做累加。

```

# 从1-10的求和
sum = 0
for x in range(1, 11): # 1-10
    sum = sum + x
print(sum)

```

结果如下：

```

D:\Python3.1
55

```

图5-4

5.2.2 while 循环语句

和 for 语句一样都能实现循环功能。而 while 循环，只要条件满足，就不断循环，条件不满足时退出循环，当然还可以和条件判断搭配来使用，例图 5-5。

```

12 # 计算100以内所有奇数之和
13 sum = 1
14 n = 99
15 while n > 0:
16     if n // 2 != 0:
17         sum = sum + n
18         n = n - 1
19     n = n - 1
20 print(sum)

```

结果如下：

```

2500

```

图5-5

第6章 用函数实现代码复用

6.1 函数的定义和调用

使用 `def` 关键字来定义函数，而函数调用返回值反馈给调用函数的语句，例图 6-2 中定义了一个计算阶乘的函数，在 `print` 语句中调用名为 `sumn()` 的函数，将计算后的结果返回给调用函数的语句。

```
# 定义函数
1 个用法
def sumn(n): # 计算阶乘
    sum = 1
    if n == 0 or n == 1:
        return 1
    for i in range(1, n + 1):
        sum = i * sum
    return sum
print(sumn(4))
```

结果如下：

```
D:\Python3.12.2\py
24
```

图6-1

6.2 函数的参数

6.2.1 默认参数

若你定义的函数拥有多个参数时，例图 6-2 当缺少了其中一个参数都会导致报错，无法运行，因此我们在缺少参数值的时候，可以在参数上添加上默认的参数以保证函数的正常运行。例图 6-3。

```

11     # 定义一个计算X^n函数
    1个用法
12  def power(x, n):  # x和n这两个参数都是位置参数
13      s = 1
14      while n > 0:
15          n = n - 1
16          s = s * x
17      return s
18
19  print(power(1))

```

结果报错如下：

```

D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
Traceback (most recent call last):
  File "D:\NAS\My code\python\test.py", line 19, in <module>
    print(power(1))
          ^^^^^^^^^
TypeError: power() missing 1 required positional argument: 'n'

```

图6-2

```

21  # 默认参数
22  # 上面的函数参数调用缺少一个参数都会导致错误
    1个用法
23  def power_default(x, n=2):  # n的参数默认为2,注意!必选参数在前,默认参数在后!
24      s = 1
25      while n > 0:
26          n = n - 1
27          s = s * x
28      return s
29
30
31  print(power_default(5))  # 当只有一个参数传入时也不会报错n的参数默认为2计算5^2

```

结果如下：

```

D:\Python3.12.2\p
25

```

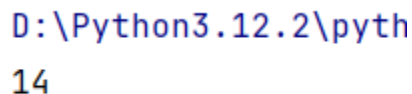
图6-3

6.2.2可变与不定长参数

当我们的参数需要多个时，可以组装一个列表（list）或者元组（tuple）类型
例如我们这里定义一个计算“ $a^2+b^2+\dots$ ”参数个数不确定的函数，如图 6-5。

```
11 # 可变参数
12 # 1个用法
13 def power_count(number): # 计算 $a^2+b^2+\dots$  参数个数不确定
14     sum = 0
15     for n in number:
16         sum=n**2+sum
17     return sum
18 print(power_count([1,2,3])) # 但是每次调用的时候必须都先组转一个list或者tuple类型
```

结果如下：



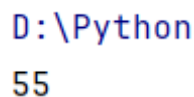
```
D:\Python3.12.2\pyth
14
```

图6-5

但图 6-5 所示的方法每次调用前必须都得组装一个列表(list)或者元组(tuple)类型比较繁琐，因此我们可以在参数面前加个“*”符号代表可变长参数类型，调用时就无需组装列表（list）或者元组（tuple）类型，例图 6-6 所示。

```
# 在参数里头加入*为可变参数
2 个用法
def power_count_no(*number):
    sum = 0
    for n in number: # 将number传过来的参数每次循环依次赋值给n
        # 即第一次循环将*number第一个参数给n, 第二次循环将*number第二个参数覆盖掉n
        sum=n**2+sum
    return sum
print(power_count_no(*number: 1,2,3,4,5)) # 调用时就无需组装list或者tuple
```

结果如下：



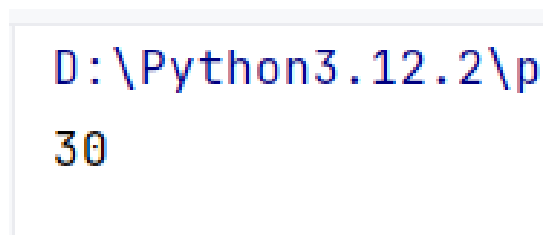
```
D:\Python
55
```

图6-6

当然在调用已经被定义的列表（list）在参数前加个*，也能把 list 或 tuple 的元素变成可变参数传进去，例图 6-7 所示。

```
21 # 在参数里头加入*为可变参数
    1个用法
22 def power_count_no(*number):
23     sum = 0
24     for n in number: # 将number传过来的参数每次循环依次赋值给n
25         #即第一次循环将*number第一个参数给n,第二次循环将*number第二个参数给n
26         sum=n**2+sum
27     return sum
28 #print(power_count_no(1,2,3,4,5)) # 调用时就无需组装list或者
29 list_num = [1,2,3,4]
30 # 在调用已经被定义的list在参数前加个*, 把list或tuple的元素变成可变参数
31 print(power_count_no(*list_num))
```

结果如下：



```
D:\Python3.12.2\p
30
```

图6-7

6.2.3 关键字参数

可变参数允许你传入 0 个或任意参数，这些可变参数在函数调用时自动组装为一个 `tuple`。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个字典（`dict`）。创建一个关键字参数只需要在参数名前面加两个 “**” 号。

例图 6-8 所示，其中除了必选参数 `name` 和 `age` 外，还接受关键字参数 `**kw`。在调用该函数时，可以只传入必选参数，`**kw` 被当作附加的功能。

```
# 关键字参数
def person(name, age, **kw): # **kw为关键字参数允许在必填参数外额外接受关键字参数
    print('name:', name, 'age:', age, 'other:', kw)

person(name='sb', age=13, gender='M', job='Engineer') # 关键字参数在函数内部自动组装为一个dict
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"
name: sb age: 13 other: {'gender': 'M', 'job': 'Engineer'}
```

图6-8

6.3 递归函数

定义：函数内部，可以调用其他函数。如果一个函数在内部间接或者直接调用自身本身，这个函数就是递归函数。例图 6-9。

```
13 # 计算n!使用递归函数调用
14 def fact(n):
15     if n==1 or n==0:
16         return 1
17     return n*fact(n-1)
18 print(fact(3)) # 函数调用通过栈数据类型实现的
19 # 每调用一次栈就会添加一层栈帧，每次返回时就减去一个栈帧，栈大小不是无限的次数过多会导致溢出
```

结果如下：

```
D:\Python3.12.2\python.exe
6
```

图6-9

6.4 匿名函数

关键字 `lambda` 表示匿名函数，匿名函数有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。因为函数没有名字，不必担心函数名冲突。格式：函数名 = [参数列表]:表达式。例图 6-10 所示。

```
f = lambda x: x * x
print(f(2))
```

结果如下：

```
D:\Python3.12.2\
4
```

图6-10

第7章 正则表达式

正则表达式的作用在于测试字符串内的特定文本，其本质用一组由字母和符号组成的“表达式”来描述一个特征，然后去验证另一个“字符串”是否符合这个特征。

作用：

1. 验证字符串是否符合指定特征，比如验证用户名或密码是否符合要求、是否是合法的邮件地址等；
2. 用来查找字符串，从一个长的文本中查找符合指定特征的字符串，比查找固定字符串更加灵活方便；
3. 用来替换，比普通的替换更强大。

7.1 正则表达式的语法规则

7.1.1 普通字符

字母、数字、汉字、下划线、以及没有特殊定义的标点符号，都是"普通字符"。表达式中的普通字符，在匹配一个字符串的时候，匹配与之相同的一个字符。

7.1.2 特殊字符

实例	描述
.	匹配除 <code>"\n"</code> 之外的任何单个字符。要匹配包括 <code>'\n'</code> 在内的任何字符，请使用象 <code>'[\n]'</code> 的模式。
<code>\d</code>	匹配一个数字字符。等价于 <code>[0-9]</code> 。
<code>\D</code>	匹配一个非数字字符。等价于 <code>[^0-9]</code> 。
<code>\s</code>	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 <code>[\f\n\r\t\v]</code> 。
<code>\S</code>	匹配任何非空白字符。等价于 <code>[^\f\n\r\t\v]</code> 。
<code>\w</code>	匹配包括下划线的任何单词字符。等价于 <code>'[A-Za-z0-9_]'</code> 。
<code>\W</code>	匹配任何非单词字符。等价于 <code>'[^A-Za-z0-9_]'</code> 。

7.1.3 限定符

表达式	可匹配
<code>{}</code>	表达式重复 <code>n</code> 次，比如： <code>"\w{2}"</code> 相当于 <code>"\w\w"</code> ； <code>"a{5}"</code> 相当于 <code>"aaaaa"</code>
<code>{m,n}</code>	表达式至少重复 <code>m</code> 次，最多重复 <code>n</code> 次，比如： <code>"ba{1,3}"</code> 可以匹配 <code>"ba"</code> 或 <code>"baa"</code> 或 <code>"baaa"</code>
<code>{m,}</code>	表达式至少重复 <code>m</code> 次，比如： <code>"\wd{2,}"</code> 可以匹配 <code>"a12"</code> , <code>"456"</code> , <code>"M12344"</code>
<code>?</code>	匹配表达式 <code>0</code> 次或者 <code>1</code> 次，相当于 <code>{0,1}</code> ，比如： <code>"a[cd]?"</code> ，可以匹配 <code>"a"</code> , <code>"ac"</code> , <code>"ad"</code> 。
<code>+</code>	表达式至少出现 <code>1</code> 次，相当于 <code>{1,}</code> ，比如： <code>"a+b"</code> 可以匹配 <code>"ab"</code> , <code>"aab"</code> , <code>"aaab"</code>

*	表达式不出现或出现任意次，相当于 {0,}，比如:"\^*b"可以匹配 "b","^^^b".....
---	---

7.1.4定位符和其他

表达式	可匹配
^	匹配输入字符串开始的位置，不匹配任何字符
\$	匹配输入字符串结尾的位置，不匹配任何字符
\b	匹配一个单词的边界，即字与空格间的位置
\B	非单词边界匹配
	左右两边表达式之间"或"关系，匹配左边或者右边，和括号配合使用

7.2正则表达式的应用

7.2.1re 模块

Python 标准库中提供 re 模块来处理正则表达式，包含所有正则表达式的功能。在 python 用 import 语句导入 re 模块即可。

7.2.2match()函数

match()方法判断是否匹配，如果匹配成功，返回一个 Match 对象，否则返回 None。语法：re.match(pattern,string,flag) ，其中 pattern:正则表达式的字符串或原生字符，string:待匹配字符串， flags:正则表达式使用时的控制标记，例图 7-1 所示。

```

11 import re
12 if re.match( pattern: r"[0-9a-z]+@[0-9a-z]+[.]+[com|xyz|net]", string: "sb@sb.xyz"):
13     print("匹配成功")
14 else:
15     print("匹配错误")

```

结果如下：

```
D:\Python3.12.2\pytho
匹配成功
```

图7-1

7.2.3search()函数

使用 search()函数来搜索字符串，找到第一个与正则表达式 pattern 匹配的位置，并返回对应的匹配对象，否则返回 None。语法：re.serch(pattern,string,flags) 其中 pattern:正则表达式的字符串或原生字符，string:待匹配字符串，flags:正则表达式使用时的控制标记，例图 7-2 所示。

```
import re
if re.search( pattern: r"very", string: "Verygood", flags=re.I): #flags=re.I时忽略大小写
    print("found")
else:
    print("not found")
```

结果如下：

```
D:\Python3.12.2\py
found
```

图7-2

7.2.4split()函数

语法：re.split(pattern, string,maxsplit, flags)。它将一个字符串按照正则表达式匹配结果进行分割，返回列表类型，maxsplit 参数限定最大分割的次数。例图 7-3 所示。

```
import re
# re.split() 分割 字符串按照表达式匹配的结果分割
string1="abc123efghigkl456mno789pqrstuvwxyz"
result=re.split( pattern: r"\d+",string1)
print(result)
result=re.split( pattern: r"\d+",string1,maxsplit=2) #maxsplit参数为最大分割次数
print(result)
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"  
['abc', 'efghigkl', 'mno', 'pqrstuvwxyz']  
['abc', 'efghigkl', 'mno789pqrstuvwxyz']
```

图7-3

7.2.5sub()函数

`re.sub(pattern,repl,string,count,flags)`，替换所有匹配正则表达式的子串，返回替换后子的字符串，`repl` 为替换匹配的字符串，`count` 为匹配最大的替换次数。

例图 7-4 所示。

```
10 import re  
11 # re.sub(pattern,repl,string,count,flags)  
12 # 替换所有匹配正则表达式的子串，返回替换后子的字符串  
13 # repl为替换匹配的字符串  
14 string1="abc123efghigkl456mno789pqrstuvwxyz"  
15 print(re.sub( pattern: r"\d+", repl: "-",string1))  
16  
17 # count为匹配最大的替换次数  
18 print(re.sub( pattern: r"\d+", repl: "-",string1,count=2))
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My code\python\test.py"  
abc-efghigkl-mno-pqrstuvwxyz  
abc-efghigkl-mno789pqrstuvwxyz
```

图7-4

第8章 面向对象编程

8.1 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

8.1.1 创建类和属性

定义类是通过 `class` 关键字，`class` 后面接着类名，类目通常用大写字母开头，紧接着是（`object`），表示该类是从哪个类继承下来的。如果没有合适的继承类，就使用 `object` 类，这是所有类最终都会继承的类。例图 8-1 所示，其中“`name`”和“`age`”共有类属性，可以类外通过实例对象或类对象访问。

2 个用法

```
class Student1: #class 名Student1
    name="sb" #公有类属性，可以在类外通过实例对象或类对象访问
    age=18

print(Student1.name,Student1.age)
```

结果如下：

sb 18

图8-1

8.1.2 实例属性

可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的“`__init__`”方法(注意：是两个下划线)，在创建实例的时候，

把我们认为必须创建的属性绑定。例图 8-2。

```
#定义__init__方法，在创建实例的时候把name和age属性绑定上去
1个用法
class Student:
    def __init__(self,name,age): # 注意__init__方法第一个参数永远是self表示创建实例本身
        self.name = name ##有了__init__方法后，不能传入空的参数，必须输入__init__方法匹配的参数
        self.age = age

s1=Student( name: "hmb", age: 18) # 创建s1实例必须填入name和age参数
print(s1.name,s1.age)
```

结果如下：

```
D:\Python3.12.2\pytho
hmb 18
```

图8-2

其中“__init__”方法第一个参数永远是 self 表示创建实例本身，有了“__init__”方法后，不能传入空的参数，必须输入 “__init__” 方法匹配的参数。

与函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 self，并且调用时不用传递该参数。除此之外，类方法与函数没用什么区别，仍然可以用函数中的参数例如默认参数、可变参数、关键字参数等。

8.1.3数据封装(类方法)

以图 8-2 为基础额外定义两个方法，可以直接在类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和类的本身是关联起来的，我们称之为类的方法。

定义类方法除了第一个参数为 `self`，要调用方法，只需要在实例变量上直接调用。例图 8-3 所示，定义一个判断等级的类方法以及打印分数的类方法，在打印分数类方法中内再调用判断等级的类方法。

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

```
# 类方法
1个用法
class Student_Methods:
    def __init__(self,name,score):
        self.name = name
        self.score=score

    1个用法
    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

    1个用法
    def print_score(self): # 定义一个打印名字和分数的方法，除了第一个参数为self，要调用一个方法，只需要在实例变量上直接调用
        print(f"您查询的名字为: {self.name} 分数为: {self.score},您的等级为: {self.get_grade()}")

sb=Student_Methods( name: "sb", score: 60) #创建实例
sb.print_score() #调用sb实例的print_score类方法
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My cc
```

```
您查询的名字为: sb 分数为: 60,您的等级为: B
```

```
进程已结束，退出代码为 0
```

图8-3

8.1.4访问限制

若想要在类中的属性不被外部访问，可以在属性名称下加上两个下划线“__”。这样就变成了私有变量，只有内部可以访问，外部不能访问。例图 8-4 所示。

类方法--访问限制

1个用法

```
class Student_Private:
```

```
    def __init__(self,name,score):
```

```
        self.__name = name # 在变量前面加__变私有变量，只有内部能访问
```

```
        self.__score = score
```

1个用法

```
    def print_score(self):
```

```
        print(f"您查询的名字为: {self.__name} 分数为: {self.__score}")
```

```
sb=Student_Private( name: "sb", score: 80)
```

```
sb.print_score()
```

结果如下：

```
D:\Python3.12.2\python.exe "D:
```

```
您查询的名字为: sb 分数为: 80
```

图8-4

如果我们强行用“`print(sb.__name)`”打印，结果是会报错的。这样就确保了外部代码不能随意修改对象内部的状态。

如果外部代码想要获取 `name` 变量或者 `score`，可以给类增加一个方法，返回对应的值给访问的函数，例图 8-5 所示。

```
## 类方法--访问限制
1个用法
class Student_Private:
    def __init__(self,name,score):
        self.__name = name # 在变量前面加__变私有变量，只有内部能访问
        self.__score = score
    def print_score(self):
        print(f"您查询的名字为: {self.__name} 分数为: {self.__score}")

#如果外部代码想要获取name变量或者score，可以给类增加一个方法，返回对应的值给访问的函数
1个用法
    def get_name(self):
        return self.__name
1个用法
    def get_score(self):
        return self.__score

sb=Student_Private( name: "sb", score: 80)
print(sb.get_name(),sb.get_score())
```

结果如下：

```
D:\Python3.
sb 80
```

图8-5

若又想要允许外部代码修改里面的一些变量，可以在类里面再定义一个方法。读者可能会认为为何不直接修改属性的变量，是因为在方法中可以对参数做检查，避免传入无效的值。例图 8-6 所示。

```
## 类方法--访问限制
1 个用法
class Student_Private:
    def __init__(self, name, score):
        self.__name = name # 在变量前面加__变私有变量，只有内部能访问
        self.__score = score
    def print_score(self):
        print(f"您查询的名字为: {self.__name} 分数为: {self.__score}")
# 如果外部代码想要获取name变量或者score，可以给类增加一个方法，返回对应的值给访问的函数
2 个用法
    def get_name(self):
        return self.__name
# 若又想要允许外部代码修改里面的一些变量，
# 可以在类里面再定义一个方法在方法中可以对参数做检查，避免传入无效的值
2 个用法
    def get_score(self):
        return self.__score
2 个用法
    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError("输出超过规定的值，请输入0-100的数值")
sb = Student_Private(name="sb", score=80)
print(sb.get_name(), sb.get_score())
sb.set_score(90) # 用set_score方法修改私有属性score的值
print(sb.get_name(), sb.get_score())
sb.set_score(200) # 超过所规定的值报错
```

结果如下：

```
Traceback (most recent call last):
  File "D:\NAS\My code\python\test\面向对象-类.py", line 63, in <module>
    sb.set_score(200) # 超过所规定的值报错
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\NAS\My code\python\test\面向对象-类.py", line 58, in set_score
    raise ValueError("输出超过规定的值，请输入0-100的数值")
ValueError: 输出超过规定的值，请输入0-100的数值
sb 80
sb 90
```

图8-6

8.2继承与多态

8.2.1继承

在面向对象程序设计中，当我们定义一个 `class` 的时候，可以从某个现有的 `class` 继承，新的 `class` 称为子类（Subclass）或称派生类，而被继承的原有的 `class` 称为基类、父类或超类（Base class、Super class）。

继承最大的好处是子类获得了父类的全部功能，向其添加新功能。例如我们新增一个父类为 `Human`，子类分别为 `Student`、`Teacher`。

如图 8-7 所示，其中由于 `Human` 定义了方法 `run()`，那么作为子类的 `Students` 自动拥有了 `run()` 方法，当子类和父类都存在相同的 `run()` 方法时，子类的 `run()` 覆盖了父类的 `run()`。

```
2 个用法
4  @ class Human: # 定义一个父类
5  @     def run(self):
6         print("激活成功! 请当社畜一辈子")
7
1 个用法
8  class Student(Human):
9         1 个用法
10         def run1(self): # 子类获得了父类的全部功能，向其添加新功能
11             print("好好读书，不然吃自己")
12
1 个用法
12 class Teacher(Human):
13  @     def run(self):
14         print("我今天要捞学生")
15
16 hxx=Student()
17 hxx.run() # 由于Human定义了方法run(), 那么作为子类的Students自动拥有了run()方法
18 hxx.run1() # 子类的新方法
19
20 t=Teacher() # 当子类和父类都存在相同的run()方法时，子类的run()覆盖了父类的run()
21 t.run()
22
```

结果如下：

```
D:\Python3.12.2\python.exe '
激活成功! 请当社畜一辈子
好好读书，不然吃自己
我今天要捞学生
```

图8-7

8.2.2多态

多态即多种形态要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个 class 的时候，我们实际上就定义了一种数据类型。例如，序列类型有多种形态：字符串、列表、元组。

在继承关系中，子类覆盖父类的同名方法，当调用同名方法的时候，系统会根据对象来判断执行哪个方法。例图 8-8 所示。

```
25 # 多态
26
27 class Dog(object):
28     def work(self):
29         pass
30
31 class ArmyDog(Dog):
32     def work(self):
33         print("追击敌人")
34
35 class DrugDog(Dog):
36     def work(self):
37         print("追查毒品")
38
39 class Person(object):
40     def work_with_dog(self, dog):
41         dog.work()
```

结果如下：

```
D:\Python3.12.2\py
追击敌人
追查毒品
```

图8-8

第9章 文件操作

9.1 文件的读写

9.1.1 读文件

要以读文件的模式打开一个文件对象，使用 Python 内置的 `open()` 函数，传入文件名和标示符。

`Open()` 函数的用法：

`Open(filename [, mode] [, encoding])`。分别为 `filename` 文件名、`mode` 文件打开模式和 `encoding` 文件编码方式。其中 `filename` 不可以省略，其他参数都可以省略，缺省时用默认值。文件打开模式如表 9-1 所示。

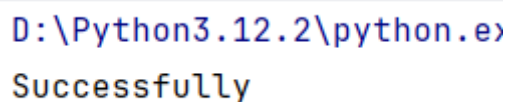
例图 9-1 图所示我们在 Python 根目录下创建一个名为 “testIO.txt” 文件，内容为 “Successfully”，标示符 'r' 表示读，这样，我们就成功地打开了一个文件。

接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python 把内容读到内存，用一个 `str` 对象表示。

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源。

```
3 # 读写文件
4 f=open('./testIO.txt','r') ##标示符'r'表示读
5 print(f.read()) #read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示
6 f.close() #关闭文件
7
```

结果如下：



```
D:\Python3.12.2\python.e>
Successfully
```

图9-1

表9-1 访问模式

模式	描述
t	文本模式 (默认)。
x	写模式，新建一个文件，如果该文件已存在则会报错。
b	二进制模式。
+	打开一个文件进行更新(可读可写)。
U	通用换行模式（不推荐）。
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
w+	打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。

a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会追加模式。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

9.1.2readline()方法

从图 9-1 中我们知道，调用 read()会一次性读取文件的全部内容，若文件太大会导致内存不够用所以保险起见我们引入一个函数“readline()”方法，逐行读取文本内容。

“readline()”的语法：

文件对象.readline()

例图 9-2 所示：

```
# readline(), 逐行读取文本内容防止文件过大
with open('./testI0.txt','r') as f:
    print(f.read())
    for line in f.readlines():_#
        print(line.strip('\n'))_# 把末尾的'\n'删掉
```

结果如下：


```
D:\Python3.12.2\pytho
Successfully
First One
Second
Third
```

图9-2

9.1.3关闭文件

基于图 9-1 的例子，我们发现每次打开文件都得要用到 `close()` 过于繁琐，为了保证每次打开文件的时候都能正确关闭文件，所以 Python 有 `with` 语句来自动帮我们调用 `close()` 方法。例图 9-3 所示代码更佳简洁，并且不必调用 “`f.close()`” 方法。

```
# 为了保证每次打开文件的时候都能正确关闭文件Python有with语句来自动帮我们调用close()方法
with open('./testI0.txt','r',encoding='utf-8') as f:
    print(f.read())
```

结果如下：

```
D:\Python3.12.2\python.e>
Successfully
```

图9-3

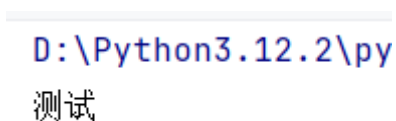
9.1.4写文件

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，从表 9-1 可知，传入标识符 'w+' 或者 'wb+' 表示写文本文件或写二进制文件。

可以反复调用 `write()` 来写入文件，但是务必要调用 `close()` 来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 `close()` 方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用 `with` 语句来得保险，例图 9-4 所示。

```
25 with open('./testIO_Write.txt', 'w+', encoding='gbk') as f:
26     # 有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘
27     f.write('测试')
28
29 with open('./testIO_Write.txt', 'r', encoding='gbk') as f:
30     print(f.read())
```

结果如下：



```
D:\Python3.12.2\py
测试
```

图9-4

9.2操作文件和目录

如果要在 Python 程序中执行这些目录和文件的操作，操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python 内置的 `os` 模块也可以直接调用操作系统提供的接口函数。在调用以下方法时得先用 “`import os`” 导入 `os` 模块。

9.2.1删除与重命名文件

要删除指定文件得要用 `os` 模块内的 `remove()`方法，假定我们当前目录下有个 `test.txt` 文件。而对于重命名来说我们要用到 `rename()`方法，例图 9-5。

```
1  import os
2  #对文件重命名:
3  os.rename(src: 'test.txt', dst: 'test.py')
4  #删掉文件:
5  os.remove('test.py')
6  |
```

图9-5

9.2.2创建和删除文件夹

操作文件和目录的函数一部分放在 `os` 模块中，创建和删除目录可以这么调用，例图 9-6。

```
7  import os
8  # 创建一个目录:
9  os.mkdir('./test')
10 # 删掉一个目录:
11 os.rmdir('./test')
```

图9-6

第10章 Python 异常处理

10.1 概念

在 Python 程序运行的过程当中，总会遇到各种各样的错误（至少）有两种错误的方式：语法错误和异常。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串。而有的错误是用户输入造成的，还有一类错误是完全无法在程序运行过程中预测的。因此 Python 内置了一套异常处理机制，来帮助我们进行错误处理。

10.2 错误（异常）处理

程序设计中经常需要考虑对各种各样的异常情况进行预判和处理，在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因，并且提示用户输入正确格式数字。

因此我们引出一个 try 机制来处理异常。

语法：

Try:

<代码块>

except<Exception Type1>:

<异常处理代码块 1>

except:

<异常处理代码块 2>

else:

<异常处理代码块 3>

finally:

<异常处理代码块 4>

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块。

例图 10-1 所示，其中“10/0”时会产生一个除法运算错误，当错误发生时，后续语句不会被执行，程序继续按照流程往下走。

```
3  try:
4      print('try...')
5      r = 10 / 0 #异常
6      print('result:', r) #异常后的代码不会执行
7  except ZeroDivisionError as e: #except由于捕获到ZeroDivisionError直接跳转至错误处理代码,即except语句块
8      print('except:', e)
9  finally:
10     print('finally...') # 执行完except后,如果有finally语句块,则执行finally语句块
```

结果如下：

```
D:\Python3.12.2\python.exe "D:\
try...
except: division by zero
finally...
```

图10-1

10.2.1python 标准异常

从例图 10-1 中，我们得知“10/0”的异常类型为“ZeroDivisionError”，当 `except<exceptions>` 由于捕获到“ZeroDivisionError”直接跳转至错误处理代码,即 `except` 语句。常见的异常类型如下：

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类

StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于 Python 解释器不是致命的)

NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告

PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

10.2.2 抛出异常（错误）

Python 可以自动引发异常，也可以通过 raise 语句显性地抛出异常，因此，错误并不是凭空产生的，而是有意创建并抛出的。我们自己编写的函数也可以抛出错误。

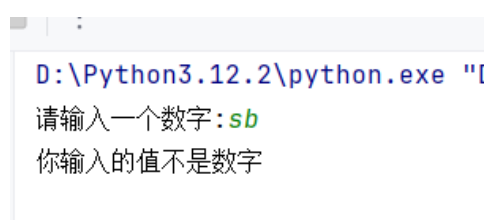
raise 的语法格式，其中 exception 名字可以在 10.2.1 中查阅：

raise <exceptionName>

例图 10-2 所示，其中“isdigit()”是检查一个字符串是否全部由数字组成，用于判断 data 是否属于数字，如果是则正常执行语句，若不是则抛出异常“ValueError”，并且执行 except 语句。

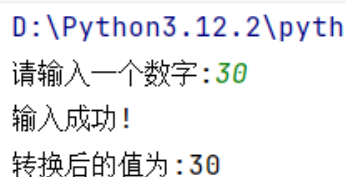
```
14  #raise 语句
15  data=input("请输入一个数字:") #字符串转换int类型
16  try:
17      if data.isdigit(): #.isdigit()检查一个字符串是否全部由数字组成
18          print("输入成功!")
19          dataint=int(data) #转换成int
20          print(f"转换后的值为:{dataint}")
21      else:
22          raise ValueError #抛出异常ValueError
23  except ValueError:
24      print("你输入的值不是数字")
```

若输入不为数字例如“sb”结果如下：



```
D:\Python3.12.2\python.exe "I
请输入一个数字: sb
你输入的值不是数字
```

若输入为数字则正常执行语句：



```
D:\Python3.12.2\pyth
请输入一个数字: 30
输入成功!
转换后的值为: 30
```

图10-2

第11章 Python 的模块使用与程序打包

11.1 模块的概念

Python 模块(Module)，是一个 Python 文件，以 .py 结尾，包含了 Python 对象定义和 Python 语句。模块让你能够有逻辑地组织你的 Python 代码段。把相关的代码分配到一个模块里能让你的代码更好用，更易懂。模块能定义函数，类和变量，模块里也能包含可执行的代码。

11.2 模块导入

在 Python 中要使用模块必须将模块进行导入。得使用 import 或者 from import 语句。

11.2.1 import 语句

import 语句语法：

```
import module1[, module2[,... moduleN]]
```

这样我们可以使用 import 语句来引入模块，例图 11-1 我们要用到模块“math”中的函数“math.fabs()”求绝对值就在 Python 文件最开始的地方用“import math”来引入“math”模块

```
import math #导入math模块
a=-10
print(int(math.fabs(a))) # math模块内的math.fabs()求绝对值
```

结果如下：

```
D:\Python
10
```

图11-1

11.2.2 from import 语句

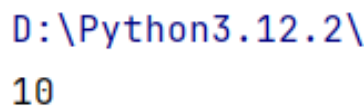
from import 语句语法:

```
from modnam import name1[,name2][,...nameN]
```

from import 语句这种导入方式不会把被导入的模块名称放在当前的字符表中,而是让你从模块中导入一个指定的部分到当前命名空间中,例我们用“from import”语句导入“math”模块的“fabs()”函数。

```
10  # from import语句
11  from math import fabs
12  a=-10
13  print(int(fabs(a))) # math模块内的fabs()求绝对值
```

结果如下:



```
D:\Python3.12.2\  
10
```

图11-2

这个声明不会把整个“math”模块导入到当前的命名空间中,它只会将“math”模块里的“fabs()”单个引入到执行这个声明的模块的全局符号表。

11.3创建自定义模块

创建自定义模块，可以将自己的一些函数功能做成一个库，不仅是你自己可以调用别人也可以调用，需要注意的是自己创建的模块需要是.py 结尾的并且在同一个文件夹下。

例我们在文件夹下创建一个 myfunction.py 的文件包含有两个函数分别是“M_abs”求绝对值和“M_sum”求和。

```
test.py  异常处理.py  模块导入.py  myfunction.py ×
1  'myfunction'
   1个用法
2  def M_abs(num): # 求绝对值
3      try:
4          if isinstance(num, int):
5              if num >= 0:
6                  print(num)
7              else:
8                  print(-num)
9          else:
10             raise ValueError
11     except ValueError:
12         print("错误，你输入的不是数字")
   1个用法
13 def M_sum(num1, num2): # 求和
14     try:
15         if isinstance(num1, int) and isinstance(num2, int):
16             print(num1 + num2)
17         else:
18             raise ValueError
19     except ValueError:
20         print("错误，你输入的不是数字")
21 > if __name__ == '__main__':
22     print("直接在自定义模块中启动myfunction") # 从自定义模块直接启动
23 elif __name__ == 'myfunction': #如果在其他地方导入该模块时，if判断将失败
24     print("外部导入myfunction")
```

在同一个文件夹内用 import 语句调用该模块：

```
18 import myfunction
19
20 myfunction.M_abs(-10)
21 myfunction.M_sum(num1: 5, num2: 9)
22
```

结果如下：

```
D:\Python3.12.2\python.exe "D:
外部导入myfunction
10
14
```

图11-3

其中“__name__”置为“__main__”（必须是左右两边各两个下划线）。如果是在模块内直接启用时“if __name__ == '__main__'”判断是 True 则执行下面的 print 语句，而如果在其他地方导入 myfunction 模块时，if 判断将失败即 False，则执行 elif 的语句。例

```
21 > if __name__ == '__main__':
22     print("直接在自定义模块中启动myfunction") # 从自定义模块直接启动
23 elif __name__ == 'myfunction': #如果在其他地方导入该模块时，if判断将失败
24     print("外部导入myfunction")
```

如果在模块内运行结果如下：

```
D:\Python3.12.2\python.exe "D:\NAS\My
直接在自定义模块中启动myfunction
```

如果在其他模块导入：

```
18 import myfunction
19
20 myfunction.M_abs(-10)
21 myfunction.M_sum(num1: 5, num2: 9)
22
```

结果如下：

```
D:\Python3.12.2\python.exe "D:
外部导入myfunction
10
14
```

图11-4

11.4程序打包

将 Python 程序打包成 exe 需要用到 pyinstaller 模块。

Pyinstaller 的安装：

在 window 下的命令提示符输入命令 `pip install pyinstaller` 即可安装成功。

11.4.1pyinstaller 的使用

进入需要打包的目录下，执行打包命令：

`Pyinstaller [opts] Filename.py`

参数	含义
-F	-onefile，打包成一个 exe
-D	-onefile，创建一个目录，包含 exe 文件，但会依赖很多文件（默认选项）
-c	-console，-noWindowed，使用控制台，无窗口（默认）
-w	-Windowed，-noconsole，使用窗口，无控制台

结语

在此篇文中，我深入探讨了 Python 编程语言的核心概念和高级特性。从基础的语法知识，到字符串处理、组合数据类型、流程控制，再到函数封装、正则表达式、面向对象编程，每一章节的学习都为我们揭示了 Python 的丰富功能和编程范式。通过对文件操作、异常处理以及模块使用的深入学习，我不仅掌握了 Python 编程的实践技能，而且对程序设计的原则和方法有了更深刻的理解。

就结论而言，Python 作为一种高效、易学且功能强大的编程语言，不仅在学术界，也在工业界得到了广泛的应用。

通过对 Python 的学习，我们不仅提升了编程能力，更重要的是，培养了解决复杂问题的逻辑思维和系统设计的能力。展望未来，Python 将继续在数据科学、人工智能、网络开发等多个领域发挥重要作用，而我们通过本研究的深入探索，为今后的大学生涯和实际应用打下了坚实的基础。

致谢

我衷心感谢所有在本文章撰写过程中给予帮助和支持的人士。

首先，我要向我的老师李妍和 B23 计科 1 班的同学表示最诚挚的感谢。在这篇文章的撰写过程中，李妍老师和同班同学给予了我宝贵的指导和建议，不仅在学习 Python 上给予我启发，也在研究方法和思路提供了极大的帮助。

特别感谢黄敏斌、黄翊豪同学，在一些我不理解的点给予了我无私的帮助，使得本文章能够顺利撰写。

此外，我还要感谢我的家人和朋友，他们在我学术道路上的鼓励和支持是我前进的动力。

最后，感谢所有未能一一列举的帮助过我的各位，正是你们的关心和支持，使我能够顺利完成这项文章。

参考文献

- [1]. GB/T 7714: 杨旭 张学义 单家凌.Python 语言程序设计基础（微课版）[M].
湖北:电子科技大学出版社,2019.
- [2]. GB/T 7714-2015: 廖雪峰. Python 教程[EB/OL]. ([2019/5/2])[2024-06-21].
<https://www.liaoxuefeng.com/wiki/1016959663602400>.
- [3]. GB/T 7714-2015: Python 基础教程[EB/OL]. (2019-01-02)[2024-06-21].
<https://www.runoob.com/python/python-tutorial.html>.