目录

[2.1.2 声明 Python 编码 1](#_Toc203733788)

[2.1.3 加入代码注释 2](#_Toc203733789)

[1．单行注释 2](#_Toc203733790)

[2．多行注释 2](#_Toc203733791)

[2.1.5 缩进代码 3](#_Toc203733792)

[2.1.7 调试 Python 代码 4](#_Toc203733793)

[任务 2.2 创建字符串变量并提取里面的数值 4](#_Toc203733794)

[2.2.1 了解 Python 变量 5](#_Toc203733795)

[2.2.2 相互转化数值型变量 6](#_Toc203733796)

[2.2.3 字符型数据的创建与基本操作 7](#_Toc203733797)

[2.2.4 任务实现 11](#_Toc203733798)

[2.3.1 掌握常用操作运算符 12](#_Toc203733799)

[Noun 15](#_Toc203733800)

## 2.1.2 声明 Python 编码

Python 3 安装后，系统默认其源码文件为 UTF-8 编码。在此编码下，全世界大多数语

言的字符都可以同时在字符串和注释中得到准确编译。

大多数情况下，通过编辑器编写的 Python 代码默认保存 UTF-8 编码脚本文件，系统通

过 Python 执行该文件时就不会出错。但是如果编辑器不支持 UTF-8 编码的文件，或者团队

合作时有人使用了其他编码格式，Python 3 就无法自动识别脚本文件，就会造成程序执行

错误，这时候对 Python 脚本文件进行编码声明就显得尤为重要了。比如，GBK 脚本文件

在没有编码声明时执行就会出错，经编码声明后，脚本文件就可以正常执行。

为源文件指定特定的字符编码，需要在文件的首行或者第二行插入一行特殊的注释行，

通常使用的编码声明格式如下。

#-\*-coding:utf-8-\*-

通过上述声明，源文件中的所有字符都会被当成 coding 指代的 UTF-8 编码对待。当然

这不是唯一的声明格式，上述格式只是普遍使用的一种形式。其他形式的声明，如

“#coding:utf-8”和“#coding=utf-8”，也都是可以的。

在编写 Python 脚本时，除了要声明编码以外，还需要注意路径声明。路径声明的格式

如下。

#!e:/Python/Python36

上述语句声明的路径为 Python 的安装路径。路径声明的目的是告诉 OS 调用

“e:/Python/Python36”目录下的 Python 解释器执行文件，一般放在脚本首行。

## 双击运行

确保已安装 Python，并且安装过程中勾选了 “Add Python to PATH” 选项。

找到任意一个.py 文件，右键点击它，选择 “属性”。

在 “常规” 选项卡中，点击 “更改” 按钮。

在弹出的窗口中，找到 Python 安装目录下的 python.exe 文件。通常路径为 C:\PythonXX\python.exe（XX 为 Python 的版本号），选择它并点击 “确定”。

返回 “属性” 窗口，确保点击了 “应用” 和 “确定” 按钮。

## 2.1.3 加入代码注释

注释对于机器编程来说是不可少的，即使是简短的几行 Python 代码，如果使用了一些

生僻的方法，那么程序开发人员也需要花一定时间才能弄明白。更何况，实际应用中常常

要面临成千上万行晦涩难懂的代码，如果对代码注释得不够彻底，时间久了恐怕连程序开

发人员自己也会弄不清代码的含义。下面将介绍 Python 注释行的用法。

Python 编程基础

28

### 1．单行注释

单行注释通常以井号（#）开头，如代码 2-1 所示。

代码 2-1 单行注释

# 这是一个单独成行的注释

>>>print("Hello,World!") # 这是一个在代码后面的注释

注释行是不会被机器编译的，在这里需要提一下的是前文介绍过的编码声明也是以#

号开头，但不属于注释行，而且编码声明需要放在首行或第二行，否则不会被机器解释。

### 2．多行注释

在实际应用中常常会有多行注释的需求，同样也可以使用#号进行注释，只需在每一行

前加#号就行。

（1）#号注释

使用#号进行注释，如代码 2-2 所示。

代码 2-2 多行注释

# 这是一个使用#号的多行注释

# 这是一个使用#号的多行注释

# 这是一个使用#号的多行注释

>>>print("Hello,World!")

显然，这种方法看起来有些笨拙。Python 中对多行注释还有另一种更加方便优雅的方

式，就是使用 3 个单引号或者 3 个双引号将注释内容括起来，达到多行或者整段内容注释

的效果。

（2）单引号注释

使用单引号注释，如代码 2-3 所示。

代码 2-3 单引号注释

'''

该多行注释使用的是 3 个单引号

该多行注释使用的是 3 个单引号

该多行注释使用的是 3 个单引号

'''

>>>print("Hello,World!")

（3）双引号注释

使用双引号注释，如代码 2-4 所示。

代码 2-4 双引号注释

"""

该多行注释使用的是 3 个双引号

该多行注释使用的是 3 个双引号

该多行注释使用的是 3 个双引号

## 2.1.4 使用多行语句

多行语句可以有两种理解：一条语句多行；一行多条语句。

一条语句多行的情况一般是语句太长，一行写完一条语句会显得很不美观，使用反斜

杠（\）可以实现一条长语句的换行，也不会被机器识别成多条语句，如代码 2-5 所示。

代码 2-5 长语句换行

>>>total = applePrice +\

... bananaPrice + \

... pearPrice

但是 Python 中，[ ]、{ }、( )里面的多行语句在换行时是不需要使用反斜杠（\）的，

如代码 2-6 所示。

代码 2-6 使用逗号换行

>>>total = [applePrice ,

... bananaPrice ,

... pearPrice]

一行多条语句，通常在短语句中应用得比较广泛。使用分号（；）可对多条短语句实现

隔离，从而在同一行实现多条语句，如代码 2-7 所示。

代码 2-7 分号实现隔离

>>>applePrice = 8; bananaPrice = 3.5; pearPrice = 5

## 2.1.5 缩进代码

Python 最具特色的就是以缩进的方式来标识代码块，不再需要使用大括号（{}），代码

看起来会更加简洁明朗。

### 1.VS Code 关键配置（确保自动缩进正常工作）

1. 打开设置（Ctrl+, 或点击左下角齿轮图标）
2. 搜索并设置以下选项：
   * **Editor: Tab Size** → 设置为4（Python 官方推荐）
   * **Editor: Insert Spaces** → 勾选（使用空格而非 Tab 字符）
   * **Editor: Auto Indent** → 选择advanced（高级自动缩进）
   * **Python > Formatting: Provider** → 选择autopep8或black（格式化工具）

同一个代码块的语句必须保证相同的缩进空格数，否则将会出错。至于缩进的空格数，

Python 并没有硬性要求，只需保证空格数一致即可。

正确缩进的 Python 代码块如代码 2-8 所示。

代码 2-8 正确缩进示例

>>>if True :

... print('我的行缩进空格数相同')

>>else :

... print('我的缩进空格数不同')

错误示范如代码 2-9 所示，最后一行的语句缩进空格数与其他行不一致，会导致代码

运行出错。

代码 2-9 错误缩进示例

>>>if True :

... print('我的行缩进空格数相同'')

>>>else :

... print('错误示范'')

... print('我的缩进空格数不同')

此外，在交互式输入复合语句时，必须在最后添加一个空行来标识结束。当代码太复

杂时，解释器将难以判断代码块从何结束，而且以空行标识结束也更便于自己进行查阅和

理解。

2.1.6 命名标识符与保留字符

标识符在机器语言中是一个被允许作为名字的有效字符串。Python 中的标识符主要用

在变量、函数、类、模块、对象等的命名中。

Python 中对标识符有如下规定。

（1）标识符可以由字母、数字和下划线组成。

（2）标识符不能以数字开头。以下划线开头的标识符具有特殊的意义，使用时需要特

别注意。

① 以单下划线开头（如\_foo）的标识符代表不能直接访问的类属性，需通过类提供的

接口进行访问，不能用“from xxx import \*”导入。

② 以双下划线开头（如\_\_foo）的标识符代表类的私有成员。

③ 以双下划线开头和结尾（如\_\_foo\_\_）的标识符是 Python 特殊方法专用的标识，如

\_\_init\_\_()代表类的构造函数。

（3）标识符字母区分大小写，例如 Abc 与 abc 是两个标识符。

（4）标识符禁止使用 Python 中的保留字。要查看某字符串是否为保留字，可以使用

iskeyword 函数。此外，使用 kwlist 函数可以查看所有保留字，如代码 2-10 所示。

你提供的代码使用了 Python 的 keyword 模块来检查字符串是否为 Python 关键字。以下是具体的使用说明：

1. **导入 keyword 模块**：该模块提供了与 Python 关键字相关的功能。
2. **调用 iskeyword() 函数**：传入一个字符串参数，判断它是否是 Python 关键字。
3. **返回值**：如果是关键字返回 True，否则返回 False。

import keyword

# 检查 "and" 是否为关键字（返回 True）

print(keyword.iskeyword("and")) # 输出: True

## 2.1.7 调试 Python 代码

程序一次写完并能正确运行的概率非常小，一般会有各种各样的 Bug 需要修正。有的

Bug 很简单，看看错误信息就知道如何解决；而有的 Bug 很复杂，需要判断出错时哪些变

量的值是正确的、哪些变量的值是错误的。因此，程序开发人员需要一整套调试程序的手

段来修复 Bug。

程序调试就是将写好的程序投入实际运行前，用手工或编译程序等方法进行测试，进

而修正语法错误和逻辑错误的过程。这是保证计算机信息系统正确性的必不可少的步骤。

写完计算机程序，必须送入计算机中进行测试，然后根据测试时所发现的错误进一步诊断，

找出原因和具体的位置并进行修正。

Python 代码可以使用 pdb（Python 自带的包）调试、Python IDE 调试（如 PyCharm）、

日志功能等进行调试。接下来介绍对于一些简单的错误怎么调试修改，如代码 2-11 所示。

>>> print "Hello,World!" # 缺少括号

SyntaxError: invalid syntax

>>> print(‘Hello,World!’) # 引号为中文引号

SyntaxError: invalid character in identifier

>>> print（‘Hello,World!’） # 括号为中文括号

SyntaxError: invalid character in identifier

## 任务 2.2 创建字符串变量并提取里面的数值

Python 基础变量主要有字符型和数值型两种，数值型变量又可分为整数、浮点数、布

尔值。创建变量时不需要声明数据类型，Python 能够自动识别数据类型。这里的任务将创

建字符串变量“Apple's unit price is 9 yuan.”，并把里面的数值提取出来，转成整型（int）

数据。

（1）创建一个字符串变量“Apple's unit price is 9 yuan.”。

（2）提取出里面的数字 9 并赋值给新的变量。

（3）查看新变量的数据类型。

（4）将提取的数字 9 转成整型（int）。

（5）确认数据类型是否转换成功。

### 2.2.1 了解 Python 变量

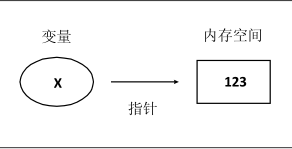
在 Python 中，变量不需要提前声明，创建时直接对其赋值即可，变量类型由赋给变量

的值决定。值得注意的是，一旦创建了一个变量，就需要给该变量赋值。

有一种“平民”的说法是，变量好比一个标签，指向内存空间的一块特定的地址。创

建一个变量时，在机器的内存中，系统会自动给该变量分配一块内存，用于存放变量值，

如图 2-1 所示。



通过 id 函数可以具体查看创建变量和变量重新赋值时内存空间的变化过程，如代码

2-12 所示。

代码 2-12 内存空间的变化

>>>x = 4

>>>id(x) # 查看变量 x 指向的内存地址

30834096L

>>>y = x # 将变量 x 重新赋给另一个新变量 y

>>>id(y)

30834096L

>>>x = 2 # 对变量 x 重新赋值

>>>x,y # 同时输出变量 x 和变量 y 的值

(2,4)

>>>id(x)

30834144L

>>>id(y)

30834096L

从代码 2-12 中可以直观地看出，一个变量在初次赋值时就会获得一块内存空间来存放

变量值。当令变量 y 等于变量 x 时，其实是一种内存地址的传递，变量 y 获得的是存储变

量 x 值的内存地址，所以当变量 x 改变时，变量 y 并不会发生改变。此外还可以看出，变

量 x 的值改变时，系统会重新分配另一块内存空间存放新的变量值。

要创建一个变量，首先需要一个变量名和变量值（数据），然后通过赋值语句将值赋给

变量。

#### 1．变量名

变量的命名须严格遵守标识符的规则，Python 中还有一类非保留字的特殊字符串（如内置函数名），这些字符串具有某种特殊功能，虽然用于变量名时不会出错，但会造成相应

的功能丧失。如 len 函数可以用来返回字符串长度，但是一旦用来作为变量名，其就失去

了返回字符串长度的功能。因此，在取变量名时，不仅要避免 Python 中的保留字，还要避

开具有特殊作用的非保留字，以避免发生一些不必要的错误，如代码 2-13 所示。

如果在一段代码中有大量变量名，而且这些变量没有错，只是取名都很随意，风格不

一，这样在解读代码时就会出现一些混淆。接下来介绍几种命名法。

（1）大驼峰（upper camel case）

所有单词的首字母都是大写，例如“MyName,YourFamily”……

大驼峰命名法一般用于类的命名。

（2）小驼峰（lower camel case）

第一个单词的首字母为小写字母，其余单词的首字母都采用大写字母，例如“myName”

“yourFamily”等。

小驼峰命名法用在函数名和变量名中的情况比较多。

#### （3）我采用\_下划线（\_）分隔

首个单词用小写字母，中间用下划线（\_）分隔后，单词的首字母为大写字母，例如

“my\_Name,your\_Family”等。

关于要使用哪种方法对变量命名，并没有统一的说法，重要的是一旦选择好了一种命

名方式，在后续的程序编写过程中一定要保持风格一致。

2．变量值

变量值就是赋给变量的数据，Python 中有 6 个标准的数据类型，分别为数字（Number）字符串（String）、列表（List）、元组（Tuple）、字典（Dictionary）、集合（Sets）。其中，

列表、元组、字典、集合属于复合数据类型。

3．变量赋值

最简单的变量赋值就是把一个变量值赋给一个变量名，只需要用等号（=）就可以实现。

同时，Python 还可以将一个值同时赋给多个变量，如代码 2-14 所示。

如果要将数字 1、2 和字符串“abc”

分别赋值给变量 a、b、c，就需使用逗号（,）隔开，如代码 2-15 所示。

代码 2-15 多个变量同时赋值

>>>a,b,c =1,2,"abc"

### 2.2.2 相互转化数值型变量

Python 3 支持的数值型数据类型有 int、float、bool、complex，Python 3 中的整数类型

int 表示长整型，没有了 Python 2 中的 long，如表 2-1 所示。

表 2-1 数值类型

数值型数据类型 中文解释 示 例

int 整数类型 10；100；1000

float 浮点数 1.0；0.11；1e-12

bool 布尔型 True；False

complex 复数 1+1j；0.123j；1+0j

int 类型指整数数值，float 类型指既有整数又有小数部分的数据类型，这些都是比较好

理解的。bool 类型只有 True（真）和 False（假）两种取值，因为 bool 继承了 int 类型，即

在这两种类型中 True 可以等价于数值 1，False 可以等价于数值 0，并且可以直接使用 bool值进行数学运算。complex 类型由实数部分和虚数部分构成，Python 中的结构形式，如 real

＋imag（J／j 后缀），实数和虚数部分都是浮点数。

在 Python 中可以实现数值型数据类型的转换，使用的内置函数有 int、float、bool、

complex。int 函数转换如代码 2-16 所示

>>>int(1.56) ; int(0.156) ; int(-1.56) ; int() # 浮点数转整型

布尔型转整型时，bool 值 True 被转成整数 1，False 被转成整数 0；复数没办法转换成整型。

bool 函数转换如代码 2-17 所示。

代码 2-17 bool 函数转换演示

bool(1) , bool(2), bool(0) # 整型转布尔型

True

True

False

从整数、浮点数、复数转布尔型的结果可以总结出一个规律：非 0 数值转布尔型都为

True，数值 0 转布尔型为 False。此外，用 bool 函数分别对空、空字符、空列表、空元组、

空字典（或者集合）进行转换时结果都为空，如果是非空，结果是 True（除去非数值 0 的

情况）。

### 2.2.3 字符型数据的创建与基本操作

相比于数值型数据，字符型数据可以理解成是一种文本，在语言领域的应用更加广泛。

Python 提供了几种方式去表达字符串，分别是使用单引号（'）、双引号（"）和三引号（'''

或者"""），下面介绍这几种方法在 Python 中的实现。

#### 1．标识字符串

（1）单引号（'）

单引号标识字符串的方法是将字符串用单引号括起来。标准 Python 库允许字符串中包

含字母、数字及各种符号。Python 3 的默认编码为 UTF-8，意味着可以在字符串中任意使

用中文也不会出错，如代码 2-18 所示。

代码 2-18 单引号标识字符串

'This is a sentence.' # 单引号标识字符串

（2）双引号（"）

双引号在字符串中的使用与单引号的用法完全相同，需要注意的是，单引号和双引号

不能混用，如代码 2-19 所示。

代码 2-19 双引号标识字符串

>>>"This is a sentence." # 双引号标识字符串

'This is a sentence.'

（3）三引号（'''或者"""）

三引号相比于单引号或者双引号，自身有一个比较特殊的功能，它能够标识一个多行

的字符串，如一段话的换行、缩进等格式都会被原封不动地保留。三引号是格式化记录一

段话的好帮手，但前后引号要保持一致，不要混用，如代码 2-20 所示。

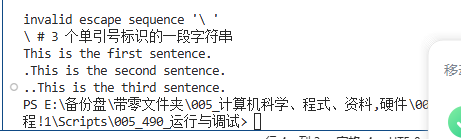
paragraph = '''\ # 3 个单引号标识的一段字符串

...This is the first sentence.

... This is the second sentence.

... This is the third sentence.'''

print(paragraph)



代码 2-20 展示了 3 个单引号标识的一段字符串，通过 print 函数打印，可以清楚地看出句子的换行和段落缩进等细节都保持了原状。另外，3 个双引号的用法一样，读者可以

动手实践。细心的读者可能会发现，代码 2-20 的命令行中有反斜杠（\），它表示字符串在

下一行继续，而不是开始一个新行。

反斜杠（\）不仅可以在字符串中担当特殊换行的角色，还可以是字符串中的转义符

#### 2．字符转义

举个简单的例子，用单引号标识一个字符串的时候，如果该字符串中又含有一个单引

号，比如“What’s happened”，Python 将不能辨识这段字符串从何处开始，又在何处结束。

此时需要用到转义符，即前文提到的反斜杠（\），使单引号只是纯粹的单引号，不具备任

何其他作用，如代码 2-21 所示。

代码 2-21 单引号转义

>>>'What's happened' # 单引号标识的字符串中含有单引号

File "<stdin>", line 1

'What's happened'

^

SyntaxError: invalid syntax

>>>'What\'s happened' # 反斜杠（\）转义单引号

"What's happened"

比较特殊的是，用双引号标识一个包含单引号的字符串时不需要转义符，但是如果其

中包含一个双引号，则需要转义。另外，反斜杠可以用来转义其本身，如代码 2-22 所示。

代码 2-22 双引号与反斜杠转义

>>>"What's happened" # 双引号标识含有单引号的字符串

"What's happened"

>>>" Double quotes(\“)" # 双引号标识的字符串里面的双引号需要转义

' Double quotes(“)'

>>>print('Backslash(\\)') # 转义反斜杠

Backslash(\)

此外，Python 中还可以通过给字符串加上一个前缀 r 或者 R 来指定原始字符串，如代

码 2-23 所示。

代码 2-23 指定原始字符串

>>>print('D:\name\python') # 以反斜杠开头的特殊字符

D:

ame\python

>>>print(r'D:\name\python') # 用 r(或者 R)指定原始字符串

D:\name\python

掌握上述介绍的方法，读者基本就可以自由地创建一个字符串了。

#### 3．字符串索引

Python 对于字符串的操作还是比较灵活的，包括字符提取、字符串切片、拼接等，但在介绍字符串操作之前，需要先掌握字符串索引的概念。

字符串索引分为正索引和负索引，通常说的索引就是指正索引。如图2-2所示，在Python

中，索引是从 0 开始的，也就是第一个字母的索引是 0，第二个索引是 1，以此类推。很明

显，正索引是从左到右去标记字母的；负索引从右到左去标记字母，然后加上一个负号（−）。

负索引的第一个值是-1，而不是-0，如果负索引的第一个值是 0，那么就会导致 0 索引指向

两个值，这种情况是不允许的。



#### 4．字符串基本操作

下面介绍提取指定位置的字符、字符串切片和字符串拼接的操作。

（1）提取指定位置的字符

Python 中只需要在变量后面使用方括号（[ ]）将需要提取的字符索引括起来，就可以

提取指定位置的字符，如代码 2-24 所示

代码 2-24 提取指定位置的字符

word = 'Python'

word[1] # 提取第二个字符

'y'

>>>word[0] # 提取第一个字符

'P'

>>>word[-1] # 提取最后一个字符

'n'

（2）字符串切片

字符串切片就是截取字符串的片段，形成子字符串。字符串切片的方式形如 s[i:j]，s

代表字符串，i 表示截取字符串的开始索引，j 代表结束索引。需要注意的是，在截取子字

符串的时候将包含起始字符，但不包含结束字符，这是一个半开闭区间。

Python 在字符串切片的功能上有很好的默认值。省略第 1 个索引，默认为 0；省略第 2

个索引，默认为切片字符串的长度，如代码 2-25 所示。

代码 2-25 字符串切片

>>>word[0:3] # 截取第 1～３个字符

'Pyth'

>>>word[:3] # 截取第１～３个字符

'Pyth'

事实上，对于这种没有意义的切片索引，Python 还是可以优雅地处理：当第 2 个索引

越界时，将被切片字符串实际长度替代；当第 1 个索引大于字符串实际长度时，返回空字

符串；当第 1 个索引值大于第 2 个索引值时，也返回空，如代码 2-26 所示。

代码 2-26 索引越界

>>>word[3:52] # 第 2 个索引越界

'hon'

>>>word[52:] # 第 1 个索引超出字符串长度

''

>>>word[-1:3] # 第 1 个索引为负，第 2 个索引正常

''

>>>word[5:3] # 第 1 个索引大于第 2 个索引

''

Python 中，字符串是不可以更改的，所以，如果给指定位置的字符重新赋值，将会出

错，如代码 2-27 所示。

代码 2-27 字符不可修改

>>>word[0] = 'p' # 字符不可被修改

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

word[0] = 'p'

TypeError: 'str' object does not support item assignment

（3）字符串拼接

如果要修改字符串，最好的办法是重新创建一个。如果只需要改变其中的小部分字符，

可以使用字符串拼接方法。

字符串拼接时，可以使用加号（＋）将两个字符串拼接起来，使用星号（\*）表示重复。

另外，相邻的两个字符串文本是会自动拼接在一起的，如代码 2-28 所示。

代码 2-28 字符串拼接

>>>'Python is' + 3 \* ' good' # 加号拼接字符串

'Python is good good good '

>>>'Python is' ' good' # 相邻字符串自然拼接

'Python is good'

如果要将字符串“Life is short,you need something.”修改成“Life is short,you need

Python.”，实现代码如代码 2-29 所示。

代码 2-29 字符串修改

sentence = 'Life is short,you need something.'

sentence[:23] + 'Python.'

'Life is short,you need Python.'

### 2.2.4 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）“Apple's unit price is 9 yuan.”中含有单引号（'），使用反斜杠（\）进行转义。

（2）直接使用方括号（[ ]）提取字符串中指定位置的字符。

（3）使用 type 函数查看数据类型。

（4）使用 int 函数将数据转换成整型。

参考代码如任务实现 2-1 所示。

任务实现 2-1

applePriceStr = 'Apple\'s unit price is 9 yuan.'

applePrice = applePriceStr[22] # 提取数值

print('提取了苹果的单价为：',applePrice,'。此刻它的数据类型为：',type(applePrice))

applePrice = int(applePrice) # 字符转数值

print('转换数据类型后：',type(applePrice))

任务 2.3 计算圆形的各参数

设计一个小程序，运用本节介绍的操作运算符实现输入、输出圆形的基本参数。

首先了解圆形的基本计算公式，如式（2-1）所示。

2

2π =π C r S r = ， （2-1）

其中，r 代表圆形的半径，C 代表圆形的周长，S 代表圆形的面积，π 是圆周率。

由式（2-1）可得式（2-2）。

2

C S

r = =

π π

（2-2）

运用运算符实现上述两个表达式。

任务分析

（1）输入半径，输出面积及周长。

（2）输入面积，输出半径及周长。

（3）输入周长，输出半径及面积。

### 2.3.1 掌握常用操作运算符

Python 中提供了一系列便利的基础运算符，可用于数据分析研究。满足基本运算需求

的运算符主要有算术运算符、比较运算符、赋值运算符。

#### 代码 2-31 比较运算符示例

>>>1 == 2 ; 1 != 2 # 数值的比较

False

True

>>>print('a' == 'b', 'a' != 'b') ; print('a' < 'b','a' > 'b') # 字母的比较

False True

True False

>>>print(ord('a'),ord('b')) # 查看字母编码

97 98

>>>print(chr(97),chr(98)) # 查看编码对应的字符

a b

>>>'# ' < '$' # 符号的比较

True

#### 1．算术运算符

算术运算符是对操作数进行运算的一系列特殊符号，能够满足一般的运算操作需求。

表 2-2 列举了 Python 3 提供的一系列算术运算符。

表 2-2 常用算术运算符

运算符 描 述 示 例

+ 加，即两个对象相加 10+20 输出结果 30

−

减，即得到负数或是一个数减去另一个数 20−10 输出结果 10

\* 乘，即两个数相乘或是返回一个被重复若干次的字符串 10\*20 输出结果 200

/ 除，即 x 除以 y 20/10 输出结果 2.0

% 取模，即返回除法的余数 23%10 输出结果 3

\*\* 幂，即返回 x 的 y 次方 2\*\*3 输出结果为 8

// 取整除，即返回商的整数部分 23//10 输出结果 2

在进行除法（/）运算时，不管商为整数还是浮点数，结果始终为浮点数。如果希望得

到整型的商，需要用到双斜杠（//）。对于其他运算，只要任一操作数为浮点数，结果就是

浮点数，如代码 2-30 所示。

代码 2-30 算数运算符示例

>>>2 / 1 ; type(2 / 1) # 单斜杠除法

2.0

<class 'float'>

>>>2 // 1 ; type(2 // 1) # 双斜杠除法

2

<class 'int'>

>>>print(1 + 2, 'and',1.0 + 2) ; print(1 \* 2, 'and',1.0 \* 2) # 加法和乘法

3 and 3.0

2 and 2.0

>>>print('23 除以 10，商为：',23 // 10,'，余数为：',23 % 10) # 商和余数

23 除以 10，商为： 2 ，余数为： 3

>>>3 \*'Python' # 字符串的 n 次重复

'PythonPythonPython'

#### 2．比较运算符

比较运算符一般用于数值的比较，也可用于字符的比较，常用比较运算符如表 2-3

所示。

当两个数值比较结果是正确时返回 True，否则返回 False。

可能关于字符的比较对于刚接触编程的读者来说会比较生疏，这里对其中的原理做简

单的介绍，感兴趣的读者也可以查找更多的资料进行深入了解。Python 中，字符是符合

ASCII 编码的，每个字符都有属于自己的编码，字符的比较本质是字符的 ASCII 编码的

比较。

表 2-3 常用比较运算符

运算符 描 述 示 例

== 等于，即比较对象是否相等 (1==2)返回 False

!= 不等于，即比较两个对象是否不相等 (1!=2)返回 True

> 大于，即返回 x 是否大于 y (1>2)返回 False

< 小于，即返回 x 是否小于 y (1<2)返回 True

>= 大于，等于即返回 x 是否大于等于 y (1>=2)返回 False

<= 小于，等于即返回 x 是否小于等于 y (1<=2)返回 True

#### 3．赋值运算符

赋值运算符用于对变量的赋值和更新，从表 2-4 中可以看出，Python 除了简单的赋值

运算符外，还有一类特殊的赋值运算符，比如加法赋值运算符、减法赋值运算符等。除简

单赋值运算符外，其他都属于特殊赋值运算符。

表 2-4 常用赋值运算符

运算符 描 述 示 例

= 简单的赋值运算符 c=a+b 将 a+b 的运算结果赋值为 c

+= 加法赋值运算符 a+=b 等效于 a=a+b

-=

减法赋值运算符 a-=b 等效于 a=a-b

\*= 乘法赋值运算符 a\*=b 等效于 a=a\*b

/= 除法赋值运算符 a/=b 等效于 a=a/b

%= 取模赋值运算符 a%=b 等效于 a=a%b

\*\*= 幂赋值运算符 a\*\*=b 等效于 a=a\*\*b

//= 取整除赋值运算符 a//=b 等效于 a=a//b

关于表 2-4 中的特殊赋值运算符，也可以将其看作是变量的快速更新，更新意味着该

变量是存在的。对于一个之前不存在的变量，则不能使用特殊赋值运算符，如代码 2-32

所示。

代码 2-32 赋值运算符示例

>>>a = 1 + 2 ; print(a) # 简单赋值运算

3

>>> print('a：',a) ; a += 4 ; print('a += 4 特殊赋值运算后，a：',a) # 特殊赋值运算

a：3

a += 4 特殊赋值运算后，a： 7

>>>f += 4 # 未定义变量不能进行特殊赋值运算

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'f' is not defined

#### 4．按位运算符

通常，数字都是使用十进制的，按位运算符会自动将输入的十进制数转为二进制数，

再进行相应的运算。

表 2-5 列举了常用按位运算符，示例中，a 为 60，b 为 13，对应的二进制值如下。

表 2-5 常用按位运算符

运 算 符 描 述 示 例

&

按位与运算符：参与运算的两个值如果相应位都为 1，则

该位的结果为 1，否则为 0

a & b 输出结果 12，

二进制值：0000 1100

|

按位或运算符：只要对应的两个二进位有一个为 1，结果

位就为 1

a | b 输出结果 61，

二进制值：0011 1101

^ 按位异或运算符：当两对应的二进位相异时，结果为 1

a ^ b 输出结果 49，

二进制值：0011 0001

~

按位取反运算符：对数据的每个二进制位取反，即把 1

变为 0，把 0 变为 1

~a 输出结果−61，

二进制值：1100 0011

<<

左移动运算符：运算数的各二进位全部左移若干位，由

“<<”右边的数指定移动的位数，高位丢弃，低位补 0

a << 2 输出结果 240，

二进制值：1111 0000

>>

右移动运算符：把“>>”左边的运算数的各二进位全部

右移若干位，“>>”右边的数指定移动的位数

a >> 2 输出结果 15，

二进制值：0000 1111

按位运算符是对二进制数的运算，其中比较难理解的应该就是取反运算，本书后面会

详细讲解相关知识。关于位运算的示例代码 2-33 所示。

代码 2-33 按位运算符示例

>>>a = 60 ; b = 13 ; print('a = 60,b =13') # 初始赋值

a = 60,b =13

>>>print('a & b =',a & b) ; print('a | b =',a | b) ; print('a ^ b =',a ^ b)

# 或、与、异位运算

a & b = 12

a | b = 61

a ^ b = 49

>>> print('~a =',~a) ; print('a << 2 =', a << 2) ; print('a >> 2 =', a >> 2)

# 取反和位移运算

~a = -61

a << 2 = 240

a >> 2 = 15

这里以按位与和按位取反运算为例，具体讲解计算过程。

（1）按位与运算

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

如下述代码所示，a 和 b 的第 3、4 位都为 1，其他位置的数都没有同时为 1，故对 a

和 b 做按位与运算的结果在第 3、4 位为 1，其余位置都为 0 时。

a = 0011 1100

b = 0000 1101

a & b = 0000 1100

（2）按位取反运算

按位取反涉及补码的计算，相对比较复杂。

十进制数的二进制原码包括符号位和二进制值。以 60 为例，其二进制原码为“0011 1100”，

从左到右第 1 位为符号位，其中，0 代表正数，1 代表负数。

对于正数来说，其补码与二进制原码相同；对于负数而言，其补码为，二进制原码符

号位保持不变，其余各位取反后再在最后一位加 1。

例 例 2-1 对 60 进行取反。

取 60 的二进制原码：0011 1100。

取补码：0011 1100。

每一位取反：1100 0011，得到最终结果的补码（负数）。

取补码：1011 1101 得到最终结果的原码。

转换为十进制数：−61，所以 60 取反后为−61。

例 例 2-2 对−61 进行取反。

取−61 的二进制原码：1011 1101。

取补码：1100 0011。

每一位取反：0011 1100，得到最终结果的补码（正数）。

取补码：0011 1100 得到最终结果的原码。

转换为十进制：60，所以−61 取反后为：60。

例 2-1 和例 2-2 已经很好地展示了正数和负数的取反操作，可以总结为以下 5 个步骤。

（1）取十进制数的二进制原码。

（2）对原码取补码。

（3）补码取反（得到最终结果的补码）。

（4）取反结果再取补码（得到最终结果的原码）。

（5）二进制原码转十进制数。

#### 5．逻辑运算符

逻辑运算符包括 and、or、not，具体用法如表 2-6 所示，示例中 a 为 11，b 为 22。

表 2-6 逻辑运算符

运 算 符 逻辑表达式 描 述 示 例

and x and y

布尔“与”，即 x and y，如果 x 为 False，返回 False；

否则它返回 y 的计算值

a and b，返回 22

or x or y

布尔“或”，即 x or y，如果 x 是 True，它返回 True；

否则它返回 y 的计算值

a or b，返回 11

not not x

布尔“非”，即 not(x)，如果 x 为 True，返回 False。

如果 x 为 False，它返回 True

not(a and b)，返回

False

逻辑运算符应用举例如代码 2-34 所示。

代码 2-34 逻辑运算符示例

>>>a = 11;b = 22;print('a = 11,b =22') # 初始赋值

a = 11,b =22

>>>print('a and b =',a and b); print('a or b =',a or b); print('not(a and b) =',

not(a and b)); # and、or、not 运算

a and b = 22

a or b = 11

not(a and b) = False

>>>a = 0;b = 22;print('a = 0,b =22') # 重新赋值

a = 0,b =22

>>>print('a and b =',a and b); print('a or b =',a or b); print('not(a and b) =',

not(a and b)); # and、or、not 运算

a and b = 0

a or b = 22

not(a and b) = True

按位运算符和逻辑运算符用于 bool 值运算时，按位&和逻辑 and 的运算效果一样，当

符号左右两个值都为 True 时，返回结果 True，否则返回 False；按位|和逻辑 or 的运算效

果一样，当符号左右两个值中有一个值为 True 时，返回结果 True，否则返回 False，如代

码 2-35 所示。

代码 2-35 bool 值运算

>>>True & True ; True and True # 按位&、逻辑 and

True

True

>>> True | False ; True or False; # 按位|、逻辑 or

True

True

>>>True & False ; True and False;

False

False

>>> False | False ; False or False;

False

False

#### 6．成员运算符

成员运算符的作用是判断某指定值是否存在于某一序列中，包括字符串、列表或元组。

成员运算符的相关解释如表 2-7 所示。

表 2-7 成员运算符

运 算 符 描 述 示 例

in

如果在指定的序列中找到值，返回 True，否则

返回 False

x in y，x 在 y 序列中，返回 True

not in

如果在指定的序列中没有找到值，返回 True，

否则返回 False

x not y，x 不在 y 序列中，返回 True

在成员运算中，对于成员的运算不仅包含值的大小，还包括类型的判断。通过代码 2-36

可以看出，在 List 中 1 是数值，所以判断数值 1 是否属于 List 时返回 True；但是判断包含

在列表中的数值 1 时，就返回结果 False，因为类型不匹配。另外，判断[4,5]是否属于 List

时，返回结果为 True，很明显是因为 List 中包含了该值。

代码 2-36 成员运算符示例

>>>List = [1,2,3.0,[4,5],'Python3'] # 初始化列表 List

>>> 1 in List # 查看 1 是否在列表内

True

>>>[1] in List # 查看[1]是否在列表内

False

>>> 3 in List # 查看 3 是否在列表内

True

>>>[4,5] in List # 查看[4,5]是否在列表内

True

>>> 'Python' in List # 查看字符串'Python'是否在列表内

False

>>>'Python3' in List # 查看字符串'Python3'是否在列表内

True

#### 7．身份运算符

身份运算符用于比较两个对象的内存地址，说明如表 2-8 所示。

表 2-8 身份运算符

运 算 符 描 述 示 例

is 用于判断两个标识符是不是引用自一个对象 x is y，如果 id(x)等于 id(y)，返回结果 1

is not 用于判断两个标识符是不是引用自不同对象

x is not y，如果 id(x)不等于 id(y)，返回

结果 1

在身份运算中，内存地址相同的两个变量进行 is 运算时，返回 True；内存地址不同的

两个变量进行 is not 运算时，返回 True。如代码 2-37 所示，当 a、b 获取一样的值时，实质

上这两个变量也就获取了同样的内存地址。

代码 2-37 身份运算符示例

>>>a = 11 ; b = 11 ; print('a = 11,b = 11') # 初始化 a、b

a = 11,b = 11

>>> a is b ; a is not b # 身份运算

True

False

>>>id(a) ; id(b) # 查看 id 地址

1347990912

1347990912

>>>a = 11 ; b = 22 ; print('a = 11,b = 22') # 重新赋值 b

a = 11,b = 22

>>> a is b ; a is not b # 身份运算

False

True

>>>id(a) ; id(b) # 查看 id

1347990912

1347991264

### 2.3.2 掌握运算符优先级

在 Python 的应用中，通常运算的形式是表达式。表达式由运算符和操作数组成。比如

1+2 就是一个表达式，“+”是操作符，“1”和“2”是操作数。

一个表达式往往不只包含一个运算符，当一个表达式存在多个运算符时，各运算符的

优先级如表 2-9 所示，处于同一优先级的运算符则从左到右依次运算。

对于表 2-9 第二行中的“+”“−”，可以更简单地理解为，放在一个数值前面，标识该

数值的正负属性。代码 2-38 展示了简单的表达式运算。

代码 2-38 运算符的优先级示例

>>>24 + 12 / 6 \*\* 2 \* 18 # 24+12/36\*18 → 24+(1/3)\*18 → 24+6

30.0

>>> 24 + 12 / ( 6 \*\* 2 ) \* 18 # 24+12/36\*18 → 24+(1/3)\*18 → 24+6

30.0

>>>24 + ( 12 / ( 6 \*\* 2 ) ) \* 18 # 24+(12/36)\*18 → 24+(1/3)\*18 → 24+6

30.0

>>>24 + ( 12 / 6 ) \*\* 2 \* 18 # 24+2\*\*2\*18 → 24+4\*18 → 24+72

96.0

>>>( 24 + 12 ) / 6 \*\* 2 \* 18 # 36/6\*\*2\*18 → 36/36\*18 → 1\*18

18.0

>>>- 4 \* 5 + 3 # -20+3

-17

>>>4 \* - 5 + 3 # -20+3

-17

### 2.3.3 任务实现

根据任务分析，本任务的具体实现过程可以参考如下操作。

（1）使用算术运算符按要求计算圆形指定参数的表达式。

（2）输入一个圆形的半径，通过表达式计算周长和面积。

（3）输入一个圆形的周长，通过表达式计算半径和面积。

（4）输入一个圆形的面积，通过表达式计算半径和周长。

（5）关于公式中的常量 pi，这里取 3.14。

（6）round 函数可以指定保留小数的位数。

参考代码如任务实现 2-2 所示。

任务实现 2-2

# -\*-coding:utf-8-\*-

'''

根据输入计算圆形的其他参数

关于圆形的相关计算公式参考正文

'''

pi = 3.14 # 设置常量

# 输入半径，求周长、面积

r = 3 # 输入圆形的半径

C = 2 \* pi \* r # 计算圆形的周长

S = pi \* r \*\* 2 # 计算圆形的面积

print('半径为',r,'的圆形，其周长等于',C,'；面积等于',S,'。')

# 输入周长，求半径、面积

C = 5 # 输入圆形的周长

r = C / ( 2 \* pi ) # 计算圆形的半径

S = pi \* r \*\* 2 # 计算圆形的面积

print('周长为'+str(C)+'的圆形，其半径为'+str(r)+'；面积等于'+str(S)+'。')

# 输入面积，求半径、周长

S = 5 # 输入圆形的面积

r = round(( S / pi ) \*\* 0.5 , 2) # 计算圆形的半径，并保留两位小数

C = round( 2 \* pi \* r , 2) # 计算圆形的周长，并保留两位小数

str\_print = '面积为' + str(S) + '的圆形，其半径为' + str(r) + '；周长等于' + str(C)

+ '。'

print(str\_print)

## 小结

Python 相比于其他编程语言，更加优雅简单，在使用中既保持了其他语言风格的基本

特点，又有自己独特的使用方法。本章介绍了 Python 的固定语法，从 Python 固定语法中

就可以发现其独特的地方。读者掌握好这部分内容，对于将来处理更复杂的程序是有很大

帮助的。

此外，本章还介绍了两个简单的数据类型，分别是数值型和字符型。下面总结一下本

章涉及的具体知识点。

（1）Python 固定语法主要体现在编码声明、注释、多行语句、行与缩进、标识符与保

留字符 5 个方面。学习 Python，固定语法是基础之基础，只有掌握了固定语法，对于后续

的知识运用、代码管理、debug 才可以得心应手。

（2）Python 变量赋值属于地址传递，两个值相同的变量事实上都指向同一地址，可以

用身份运算符进行检验。

（3）Python 3 的数据类型包含了 4 个子类，分别是 int、float、bool、complex。每种子

类之间相互联系，又互相独立。

（4）Python 3 的字符类型可由单引号、双引号、三引号（3 个单引号或者 3 个双引号）

进行标识。同时，对于字符串的索引取值、切片、拼接的操作，Python 3 都有自己内置的

方法可以简单操作。

（5）Python 一共有 7 种类型的操作符，是算术运算符、比较运算符、赋值运算符、按

位运算符、逻辑运算符、成员运算符和身份运算符。一个表达式可以包含一个或多个运算

符，这时需要严格遵守运算符的优先级进行运算。

# 数组

在Python里，数组一般指的是`list`或者`numpy`库的`array`。下面为你介绍它们的基本用法。

## ### 1. Python内置列表（list）

这是Python最常用的数组形式，其特点是可以存储不同类型的元素，并且支持动态调整大小。

\*\*创建列表\*\*

```python

# 创建空列表

empty\_list = []

# 创建带有初始元素的列表

numbers = [1, 2, 3, 4, 5]

mixed = [1, "hello", True, 3.14]

nested = [[1, 2], [3, 4], [5, 6]]

```

\*\*访问元素\*\*

```python

# 索引从0开始

print(numbers[0]) # 输出：1

print(numbers[-1]) # 输出：5（-1表示最后一个元素）

# 修改元素

numbers[0] = 10

print(numbers) # 输出：[10, 2, 3, 4, 5]

```

\*\*常用操作\*\*

```python

# 添加元素

numbers.append(6) # 在末尾添加元素

numbers.insert(2, 2.5) # 在指定位置插入元素

# 删除元素

numbers.remove(2.5) # 删除指定值的元素

numbers.pop() # 删除并返回最后一个元素

# 切片操作

print(numbers[1:3]) # 输出：[2, 3]

print(numbers[:-1]) # 输出：[10, 2, 3, 4, 5]

# 其他常用方法

print(len(numbers)) # 获取长度

print(numbers.index(3)) # 获取元素索引

print(numbers.count(2)) # 统计元素出现次数

numbers.sort() # 排序

numbers.reverse() # 反转

```

### 2. NumPy数组

`numpy`数组是专门用于高效数值计算的数组，它的元素类型必须一致，并且支持向量化操作。

\*\*安装NumPy\*\*

```bash

pip install numpy

```

\*\*创建NumPy数组\*\*

```python

import numpy as np

# 从列表创建数组

arr1 = np.array([1, 2, 3, 4, 5])

arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# 创建特殊数组

zeros = np.zeros((3, 4)) # 创建3行4列的全0数组

ones = np.ones((2, 3)) # 创建2行3列的全1数组

range\_arr = np.arange(10) # 创建0到9的数组

linspace = np.linspace(0, 1, 5) # 创建0到1之间的5个等距点

```

\*\*数组属性和访问\*\*

```python

print(arr2.shape) # 输出：(2, 3)，表示2行3列

print(arr2.ndim) # 输出：2，表示二维数组

print(arr2.dtype) # 输出：int64，表示元素类型

# 索引和切片

print(arr2[0, 1]) # 输出：2，访问第1行第2列的元素

print(arr2[:, 1]) # 输出：[2 5]，获取第2列的所有元素

```

\*\*向量化操作\*\*

```python

# 数学运算

a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

print(a + b) # 输出：[5 7 9]

print(a \* b) # 输出：[4 10 18]

print(a \*\* 2) # 输出：[1 4 9]

# 常用函数

print(np.sum(a)) # 求和

print(np.mean(a)) # 求平均值

print(np.max(a)) # 求最大值

print(np.sin(a)) # 正弦函数

```

### 3. 列表和NumPy数组的主要区别

| \*\*特性\*\* | \*\*列表（list）\*\* | \*\*NumPy数组\*\* |

|----------------|---------------------------------|--------------------------------|

| \*\*元素类型\*\* | 可以是不同类型 | 必须是相同类型 |

| \*\*内存效率\*\* | 较低 | 较高 |

| \*\*运算速度\*\* | 较慢（循环操作） | 快（向量化操作） |

| \*\*多维操作\*\* | 需要嵌套列表，操作较复杂 | 支持多维数组，操作简洁 |

| \*\*科学计算\*\* | 不适合 | 专为科学计算设计 |

### 选择建议

- 如果你需要存储不同类型的数据，或者进行简单的增删改查操作，使用Python内置列表。

- 如果你需要进行大量数值计算（如矩阵运算、统计分析等），使用NumPy数组。

# 三函数

## 3.1print(f"\n[调试信息]")

print() 是 Python 的打印函数，用于输出信息到控制台

f"\n[调试信息]" 是一个 f-string 格式的字符串

f-string（格式化字符串字面值），是 Python 3.6 及以上版本引入的一种字符串格式化方法。

它的作用是让字符串中可以直接嵌入变量或表达式，语法是在字符串前加 f，然后用 {} 包裹变量或表达式，

\n 表示换行符，会在打印内容前先换行

[调试信息] 是实际要打印的内容，通常用于标识这是调试输出

# 四关键字

try 是异常处理机制的核心关键字之一，用于包裹可能会出现错误（异常）的代码块。

它的基本用法是与 except 配合，形成 try...except 结构：

作用说明：

程序会先执行 try 块中的代码

如果 try 块中代码正常运行，不出现错误，则跳过 except 块，继续执行后面的代码

如果 try 块中代码出现错误（抛出异常），则立即停止执行 try 块，转而执行 except 块中的错误处理代码

# 四Python扩展

## 3.1 ptthon

### 3.1.1激活事件

1. onDebugInitialConfigurations  
   当调试初始配置（比如创建 / 加载 Python 调试相关的 launch.json 配置）时，触发扩展激活。
2. onLanguage:python  
   检测到当前编辑文件是 Python 语言（文件后缀 .py 等）时，激活扩展，是最基础的语言关联激活条件。
3. onDebugResolve:python  
   调试解析阶段（比如确认 Python 调试器、解析调试参数）时，激活扩展，保障调试功能可用。
4. onCommand:python.copilotSetupTests  
   执行 python.copilotSetupTests 命令（比如用 Copilot 辅助生成测试代码的命令）时，激活扩展。
5. workspaceContains:xxx 系列  
   检测到工作区（项目目录）包含指定文件时激活，比如：
   * mspythonconfig.json / pyproject.toml：识别 Python 项目配置文件（常见于项目工程化、依赖管理场景）。
   * setup.py / requirements.txt：识别 Python 传统项目（setup.py 用于打包，requirements.txt 用于依赖声明 ）。
   * \*.py：工作区有任意 Python 脚本文件时激活，覆盖基础开发场景。
   * .venv：检测到虚拟环境目录（Python 常用虚拟环境管理方式），激活以适配虚拟环境的工具链（如解释器路径、依赖）。

#### 1. 工作区文件检测类

* workspaceContains:.conda  
  当工作区（项目目录）包含 .conda 文件时激活扩展，.conda 常见于 Conda 环境配置场景，用于关联 Conda 环境管理。
* workspaceContains:venv  
  工作区存在 venv 目录（Python 官方虚拟环境默认名称）时激活，适配虚拟环境开发需求。

#### 2. Language Model Tool 命令类

这一系列事件与 VS Code 语言模型工具（Language Model Tool）的 Python 环境操作强相关，触发场景是**执行对应命令时激活扩展**，覆盖 Python 环境管理全流程：

* onLanguageModelTool:get\_python\_environment\_details  
  执行 “获取 Python 环境详情” 命令（比如查看解释器路径、依赖）时激活，用于环境信息查询。
* onLanguageModelTool:get\_python\_executable\_details  
  执行 “获取 Python 可执行文件详情” 命令（定位 python.exe/python3 路径、版本）时激活，辅助调试环境问题。
* onLanguageModelTool:install\_python\_packages  
  执行 “安装 Python 包” 命令（比如通过 UI / 命令行装库）时激活，保障依赖安装功能可用。
* onLanguageModelTool:configure\_python\_environment  
  执行 “配置 Python 环境” 命令（切换解释器、设置环境变量）时激活，适配不同项目环境。
* onLanguageModelTool:create\_virtual\_environment  
  执行 “创建虚拟环境” 命令（新建 venv/conda 虚拟环境）时激活，覆盖环境初始化场景。

### 命令

这些是 VS Code 中 Python 扩展提供的命令，以下是红框中各个命令的使用方式：

### 1. `python.clearWorkspaceInterpreter`（清除工作区解释器设置）

\*\*使用场景\*\*：当你想要清除当前工作区中已经配置的 Python 解释器，重新进行配置时使用。

\*\*操作步骤\*\*：

- 打开 VS Code 的命令面板 ，可以通过快捷键 `Ctrl + Shift + P`（Windows 和 Linux 系统）或 `Command + Shift + P`（Mac 系统）打开。

- 在命令面板中，输入 `python.clearWorkspaceInterpreter` ，然后按下回车键，此时工作区中已配置的 Python 解释器设置就会被清除，之后你可以通过 `Python: Select Interpreter` 命令重新选择合适的 Python 解释器。

### 2. `python.configureTests`（配置测试）

\*\*使用场景\*\*：当你需要为 Python 项目配置测试框架（如 `unittest`、`pytest` 等）的相关设置时使用。

\*\*操作步骤\*\*：

- 同样通过快捷键打开命令面板，输入 `python.configureTests` 并回车。

- 此时会弹出一个选项列表，让你选择项目所使用的测试框架 ，选择对应的测试框架后，VS Code 会根据所选框架进行相关的配置，比如识别测试文件和测试函数等。配置完成后，你可以使用 VS Code 的测试运行功能来运行测试。

### 3. `python.createEnvironment`（创建环境）

\*\*使用场景\*\*：在开发 Python 项目时，为了隔离不同项目的依赖，你可以使用此命令创建虚拟环境，如 `venv` 或 `conda` 环境。

\*\*操作步骤\*\*：

- 打开命令面板，输入 `python.createEnvironment` 并回车。

- 接着会提示你选择创建环境的类型，比如 `venv` 或者 `Conda` 等 。

- 选择好环境类型后，按照提示选择环境的创建路径等相关设置，VS Code 就会自动创建对应的虚拟环境，创建完成后，你可以将项目配置为使用这个新创建的虚拟环境。

### 4. `python.createEnvironment-button`（创建环境按钮，功能与 `python.createEnvironment` 类似）

\*\*使用场景\*\*：同样用于创建 Python 虚拟环境。

\*\*操作步骤\*\*：

- 一般在 VS Code 的 Python 相关视图（比如状态栏的 Python 解释器选择区域附近 ）中，会有一个图形化的按钮，点击这个按钮（如果有显示的话），就会触发创建环境的流程，后续操作和 `python.createEnvironment` 命令类似，即选择环境类型和相关设置来创建虚拟环境。

### 5. `python.createNewFile`（新建 Python 文件）

\*\*使用场景\*\*：当你想要在当前项目中快速新建一个 Python 代码文件时使用。

\*\*操作步骤\*\*：

- 打开命令面板，输入 `python.createNewFile` 并回车。

- 此时会弹出一个输入框，让你输入新建 Python 文件的名称（包含 `.py` 后缀），输入名称后回车，VS Code 就会在当前工作区中创建一个新的 Python 文件，并自动打开编辑器供你编写代码。

### 设置

红框中的设置是 VS Code 中 Python 扩展的相关配置项，以下是它们的使用方法：

打开 VS Code 的设置界面，可以通过菜单栏的 “文件” -> “首选项” -> “设置” 进入，或者使用快捷键 Ctrl + ,

在设置界面中，搜索设置ID例如“python.analysis.aiCodeActions”

### 1. `python.activeStateToolPath`

\*\*含义\*\*：指定 ActiveState 运行时状态工具可执行文件的路径（版本 0.36+）。ActiveState 是一个用于管理 Python 运行时环境和包的工具。

\*\*使用场景\*\*：当你使用 ActiveState 来管理 Python 环境，且 VS Code 需要定位其相关工具来执行操作（如激活环境、安装包等）时，需要配置此路径。

\*\*设置方法\*\*：

- 打开 VS Code 的设置界面，可以通过菜单栏的 “文件” -> “首选项” -> “设置” 进入，或者使用快捷键 `Ctrl + ,`（Windows 和 Linux 系统） 或 `Command + ,`（Mac 系统）。

- 在设置界面中，搜索 `python.activeStateToolPath`。

- 在对应的输入框中填入 ActiveState 工具可执行文件的完整路径，例如 `C:\Program Files\ActiveState\bin\activate.exe`（Windows 示例路径，实际根据安装情况而定）。

### 2. `python.autoComplete.extraPaths`

\*\*含义\*\*：该设置用于指定需要由自动完成引擎导入的额外路径列表。当你的 Python 项目依赖一些自定义模块或者第三方库，且这些库的路径不在系统默认的搜索路径中时，通过添加这些路径，能让 VS Code 的代码自动补全功能识别并使用这些模块。

\*\*使用场景\*\*：比如你有一个自定义的 Python 模块放在某个特定目录下，或者使用了一些没有通过常规 `pip install` 方式安装到系统默认库路径的包，这时就需要将这些模块或包所在的目录路径添加到该列表中，以实现代码自动补全。

\*\*设置方法\*\*：

- 进入 VS Code 的设置界面。

- 搜索 `python.autoComplete.extraPaths`。

- 点击输入框，以数组形式添加路径。例如，在 Windows 系统中，如果你有一个自定义模块在 `C:\my\_custom\_modules` 目录下，那么可以写成 `["C:\\my\_custom\_modules"]`；在 Linux 或 Mac 系统中，假设路径是 `/home/user/my\_modules`，则写成 `["/home/user/my\_modules"]` 。

### 3. `python.condaPath`

\*\*含义\*\*：用于指定要激活的 Conda 可执行文件的路径（版本 4.4+）。Conda 是一个常用的 Python 环境管理工具，通过配置此路径，VS Code 可以找到 Conda 并使用它来管理 Python 环境，如创建、激活、删除环境等操作。

\*\*使用场景\*\*：当你使用 Conda 管理 Python 项目的环境，并且希望在 VS Code 中直接操作 Conda 环境时，需要配置该路径。

\*\*设置方法\*\*：

- 打开 VS Code 的设置界面。

- 搜索 `python.condaPath`。

- 在输入框中填入 Conda 可执行文件的完整路径。在 Windows 系统中，默认路径可能是 `C:\Users\<username>\Anaconda3\Scripts\conda.exe` ；在 Linux 或 Mac 系统中，可能是 `/home/<username>/anaconda3/bin/conda` （具体路径根据实际安装情况而定）。

### 4. `python.createEnvironment.contentButton`

\*\*含义\*\*：这个设置用于控制在编辑器中是否显示或隐藏“requirements.txt”或其他相关文件的创建环境按钮 。

\*\*使用场景\*\*：如果你希望在 VS Code 中创建 Python 环境时，能便捷地通过按钮操作生成 `requirements.txt` 等文件来管理项目依赖，或者不希望显示这个按钮，可以通过此设置进行调整。

\*\*设置方法\*\*：

- 进入 VS Code 的设置界面。

- 搜索 `python.createEnvironment.contentButton`。

- 它是一个布尔值类型的设置，勾选表示显示按钮，取消勾选表示隐藏按钮 。

## 3.2Python Debugger

这些是 \*\*VS Code Python Debugger 扩展\*\* 提供的调试命令，用于控制 Python 代码调试流程，以下是红框内命令的详细作用：

### 1. `debugpy.clearCacheAndReload`

- \*\*作用\*\*：清除调试器缓存并重新加载窗口。

- \*\*场景\*\*：调试中遇到缓存冲突（比如断点不更新、旧代码逻辑残留）时，执行此命令可强制重置调试环境，让新代码/配置生效。

### 2. `debugpy.debugInTerminal`

- \*\*作用\*\*：在终端中调试 Python 文件（直接运行 + 附加调试器）。

- \*\*场景\*\*：快速调试单个脚本，无需复杂 `launch.json` 配置。执行后，VS Code 会在集成终端启动 Python 调试会话，支持断点、变量查看。

### 3. `debugpy.debugUsingLaunchConfig`

- \*\*作用\*\*：通过 `launch.json` 配置启动调试。

- \*\*场景\*\*：调试复杂项目（多文件、自定义参数、环境变量）时，先在 `launch.json` 写好调试配置（如指定解释器、传参、模拟生产环境），执行此命令即可按配置启动调试，覆盖高级调试需求。

简单说，这些命令覆盖了 \*\*调试环境重置\*\*（`clearCacheAndReload`）、\*\*快速脚本调试\*\*（`debugInTerminal`）、\*\*复杂项目调试\*\*（`debugUsingLaunchConfig`）的全流程，适配不同调试场景 。

## 3.3PyFormat

pyformat.sortImports

### 一、命令解释

红框里的 `pyformat.sortImports` 是 PyFormat 扩展在 VS Code 里提供的命令，作用是\*\*对 Python 代码中的导入语句（`import` 相关代码 ）进行排序整理\*\* ，让导入部分的代码更规范、整洁，符合常见的代码风格（类似 `isort` 工具的功能，自动梳理导入顺序 ）。

### 二、使用方法

#### 方式 1：命令面板调用（常用）

- 打开 VS Code 中要处理的 Python 文件。

- 按 `Ctrl+Shift+P`（Windows、Linux）或 `Command+Shift+P`（Mac）调出\*\*命令面板\*\* 。

- 在命令面板输入 `PyFormat: Sort Imports`（对应 `pyformat.sortImports` 功能），回车执行，就能自动整理当前文件的导入语句排序。

#### 方式 2：绑定快捷键（自定义高效）

- 点击 VS Code 左下角设置图标 → 选“键盘快捷方式” 。

- 搜索框输入 `pyformat.sortImports` 找到对应命令，点击旁边 “+” 号，设置你习惯的快捷键（比如 `Ctrl+Alt+I` ），之后在 Python 文件里按设定快捷键，直接触发排序。

#### 方式 3：右键菜单（部分场景）

如果扩展配置了菜单上下文关联（看截图里“菜单上下文”是 `commandPalette` ，可能默认没直接右键入口，若后续扩展更新支持，可在编辑器右键菜单找对应排序选项点击 ）。

简单说，就是借助这个命令/功能，让 Python 代码里的 `import` 部分自动规整，让代码更美观、规范，减少手动整理导入的麻烦 。

## 3.4Pylance

设置ID说明

默认值

python，analysis.addHoverSummaries启用/禁用在悬停工具提示中显示助手所生成摘要的A1协助。

false python.analysis.aiCodeActions启用/禁用AI辅助的代码操作。需要启用Copilot Chat扩展。

"implementAbstractc]python，analysis，autoFormatstrings When typing a（in a string，automatically prefix the string with an f

false

python.analysis，autoImportCompletions启用自动导入补全。

false

python，analysis，autolndent键入Python 代码时，根据语言语义自动调整缩进。

true

python，analysis，autoSearchPaths自动添加常用搜索路径，例如src.true python.analysis.autoSplitStrings拆分字符串时自动添加引号和行延续字符。

true python，analysis.cacheLSPData缓存LSP数据以加快完成速度。

false

[实验性」可能导致Pylance停止工作。

# 四实战例

for i in range(1, 10):

row = [f"{j}×{i}={i\*j}" for j in range(1, i+1)]

print('\t'.join(row))

这段代码使用了 Python 的循环、列表推导式和字符串方法来生成九九乘法表。下面是对代码的详细解释：

### \*\*1. 外层循环：`for i in range(1, 10)`\*\*

- \*\*功能\*\*：遍历数字 1 到 9，控制乘法表的行数（即当前处理的乘数）。

- \*\*`range(1, 10)`\*\*：生成从 1 到 9 的整数序列（不包含 10）。

### \*\*2. 列表推导式：`[f"{j}×{i}={i\*j}" for j in range(1, i+1)]`\*\*

- \*\*功能\*\*：生成当前行的所有乘法表达式（例如，当 `i=3` 时，生成 `["1×3=3", "2×3=6", "3×3=9"]`）。

- \*\*语法拆解\*\*：

- \*\*`f"{j}×{i}={i\*j}"`\*\*：使用 f-strings 格式化字符串，将变量 `j`（被乘数）、`i`（乘数）和乘积 `i\*j` 拼接成表达式。

- \*\*`for j in range(1, i+1)`\*\*：遍历从 1 到 `i` 的数字，控制每行的列数（例如，第 3 行有 3 个表达式）。

### \*\*3. 字符串连接：`'\t'.join(row)`\*\*

- \*\*功能\*\*：将列表 `row` 中的所有字符串用制表符 `\t` 连接成一个完整的行。

- 例如，`["1×3=3", "2×3=6", "3×3=9"]` → `"1×3=3\t2×3=6\t3×3=9"`。

- \*\*`join()`\*\*：字符串方法，用于将列表中的元素按指定分隔符连接成字符串。

### \*\*4. 执行流程示例\*\*

- 当 `i=1` 时：

- `row = ["1×1=1"]`

- `print('\t'.join(row))` → 输出 `1×1=1`

- 当 `i=2` 时：

- `row = ["1×2=2", "2×2=4"]`

- `print('\t'.join(row))` → 输出 `1×2=2 2×2=4`

- 以此类推，直到 `i=9`。

### \*\*输出效果\*\*

```

1×1=1

1×2=2 2×2=4

1×3=3 2×3=6 3×3=9

1×4=4 2×4=8 3×4=12 4×4=16

1×5=5 2×5=10 3×5=15 4×5=20 5×5=25

1×6=6 2×6=12 3×6=18 4×6=24 5×6=30 6×6=36

1×7=7 2×7=14 3×7=21 4×7=28 5×7=35 6×7=42 7×7=49

1×8=8 2×8=16 3×8=24 4×8=32 5×8=40 6×8=48 7×8=56 8×8=64

1×9=9 2×9=18 3×9=27 4×9=36 5×9=45 6×9=54 7×9=63 8×9=72 9×9=81

```

### \*\*关键语法总结\*\*

1. \*\*列表推导式\*\*：简洁地创建列表的语法（`[表达式 for 变量 in 序列]`）。

2. \*\*f-strings\*\*：Python 3.6+ 支持的字符串格式化方式（`f"文本{变量}文本"`）。

3. \*\*字符串 `join()` 方法\*\*：将列表元素连接为字符串（`分隔符.join(列表)`）。

通过组合这些语法，代码用短短两行就实现了完整的九九乘法表输出。

# Noun

|  |  |  |  |
| --- | --- | --- | --- |
| False | [fɔːls] | 错误的 |  |
| print | /prɪnt/ | 打印 |  |
| shutil | 舒蒂尔 |  |  |
| shell | shell | [ʃel] | 壳 |
|  | utilities | [ju(ː)ˈtɪlɪtiz] | 实用 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Shutil

在 Python 中，`import shutil` 语句用于导入 \*\*`shutil` 模块\*\*（全称为 "shell utilities"）。这个模块提供了一系列高级文件操作功能，特别是针对文件和目录的复制、移动、删除等操作，比 `os` 模块的文件操作更强大、更便捷。

### `shutil` 模块的核心功能

#### 1. \*\*文件复制\*\*

- \*\*`shutil.copy(src, dst)`\*\*：复制文件内容和权限，但不保留其他元数据（如修改时间）。

- \*\*`shutil.copy2(src, dst)`\*\*：复制文件内容、权限和元数据（如修改时间、文件所有者等），功能最完整。

- \*\*`shutil.copyfile(src, dst)`\*\*：仅复制文件内容，不复制权限和元数据。

\*\*示例\*\*：

```python

import shutil

# 复制文件并保留元数据

shutil.copy2('source.txt', 'destination.txt')

```

#### 2. \*\*目录递归复制\*\*

- \*\*`shutil.copytree(src, dst, dirs\_exist\_ok=False)`\*\*：递归复制整个目录树（包括所有子目录和文件）。

- \*\*`dirs\_exist\_ok=True`\*\*：允许目标目录已存在（Python 3.8+ 支持）。

\*\*示例\*\*：

```python

import shutil

# 递归复制目录（目标目录不能已存在，除非设置 dirs\_exist\_ok=True）

shutil.copytree('source\_dir', 'destination\_dir', dirs\_exist\_ok=True)

```

#### 3. \*\*文件/目录删除\*\*

- \*\*`shutil.rmtree(path)`\*\*：递归删除目录及其所有内容（相当于 Linux 的 `rm -r` 命令）。

- \*\*注意\*\*：此操作不可逆，需谨慎使用！

\*\*示例\*\*：

```python

import shutil

# 删除目录及其所有内容

shutil.rmtree('directory\_to\_delete')

```

#### 4. \*\*文件移动/重命名\*\*

- \*\*`shutil.move(src, dst)`\*\*：移动文件或目录，相当于文件系统中的“剪切-粘贴”操作。

- 如果目标路径与源路径在同一目录下，相当于重命名。

\*\*示例\*\*：

```python

import shutil

# 移动文件到新目录

shutil.move('old\_path/file.txt', 'new\_path/file.txt')

# 重命名文件

shutil.move('old\_name.txt', 'new\_name.txt')

```

#### 5. \*\*磁盘空间查询\*\*

- \*\*`shutil.disk\_usage(path)`\*\*：返回指定路径的磁盘使用情况（总空间、已用空间、可用空间）。

\*\*示例\*\*：

```python

import shutil

usage = shutil.disk\_usage('/') # Linux/macOS

# 或 usage = shutil.disk\_usage('C:\\') # Windows

print(f"总空间: {usage.total / (1024\*\*3):.2f} GB")

print(f"已用空间: {usage.used / (1024\*\*3):.2f} GB")

print(f"可用空间: {usage.free / (1024\*\*3):.2f} GB")

```

### 为什么需要 `shutil`？

- \*\*相比 `os` 模块\*\*：`os` 模块提供的是底层文件操作（如 `os.remove()` 删除文件、`os.rename()` 重命名文件），而 `shutil` 提供更高级的功能（如递归复制、保留元数据等）。

- \*\*相比直接调用系统命令\*\*：`shutil` 是跨平台的，无需担心 Windows、Linux、macOS 之间的命令差异。

### 总结

`shutil` 模块是 Python 处理文件和目录操作的“瑞士军刀”，适合处理复杂的文件管理任务。例如，在你的代码中使用 `shutil.copy2()` 可以安全地复制文件并保留其所有元数据，避免手动处理文件权限和时间戳的麻烦。