速通系列

秒学 Python

(语法入门)2026

Download URL

PUBLISH ID: 43251224001
INFORMATION QUERY

主编：戴君豪

1 Ethernos Publisher

目录

1 基本入门 4
1.1 Python 环境的下载与安装 4
1.1.1 下载 Python 4
1.1.2 安装 Python 5
1.1.3 验证安装 6
1.2 你的第一个 python 程序 7
1.3 变量，数据类型，输入 8
1.3.1 变量 8
1.3.2 数据类型 12
1.3.3 输入 19
1.4 数据处理 24
1.4.1 算术运算符 24
1.4.2 赋值运算符 27
1.4.3 关系运算符 32
1.4.4 逻辑运算符 37
1.5 分支结构 42
1.5.1 分支语句 42
1.5.2 if 实现单分支 42
1.5.3 if 的嵌套 44
1.5.4 if 实现多分支 46
1.5.5 分支结构实践 49
1.6 循环结构 52
1.6.1 为什么需要循环 52
1.6.2 while 循环 52
1.6.3 for 循环与 range() 54
1.6.4 break 和 continue 56
1.6.5 循环嵌套 58
1.6.6 循环综合实践 60
1.7 列表 63
1.7.1 什么是列表 63
1.7.2 列表的索引与访问 63
1.7.3 列表的切片操作 64
1.7.4 修改列表元素 65

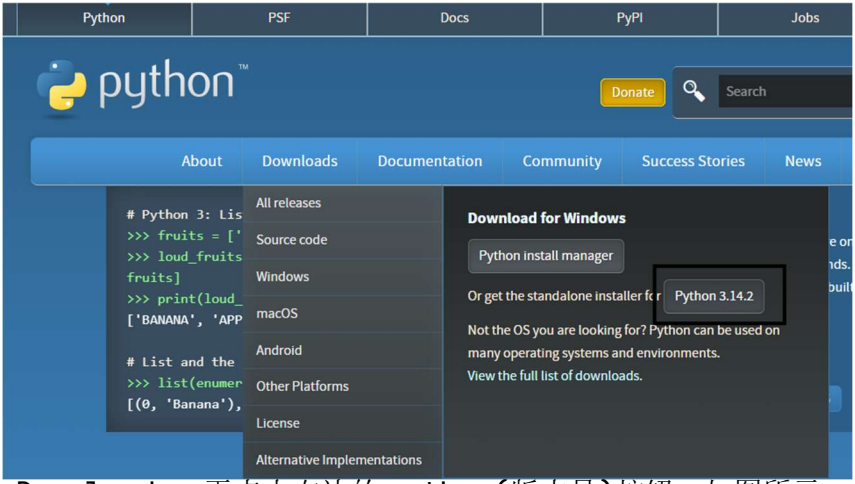
1.7.5 列表常用方法（增删操作） 66
1.7.6 列表的其他实用操作 67
1.7.7 *列表推导式（选学） 68
1.7.8 循环与列表的综合应用 69
2 语法进阶 72
2.1 函数基础 72
2.1.1 为什么需要函数 72
2.1.2 函数的定义与调用 72
2.1.3 函数的参数：让函数更灵活 74
2.1.4 函数的返回值：把结果送回来 75
2.1.5 变量的作用域：你的变量在哪有效？ 77
2.1.6 函数综合实践 78
2.2 函数进阶 81
2.2.1 *匿名函数 lambda 81
2.2.2 *高阶函数：函数当参数用 82
2.2.3 递归函数：自己调用自己 83
2.2.4 函数也是对象 84
2.2.5 闭包：函数记住外部状态 85
2.2.6 *装饰器初探（选学） 86
2.2.7 函数进阶综合实践 87

1 基本入门

1.1 Python 环境的下载与安装

1.1.1 下载 Python

第一步：访问 Python 官网：python.org(或用二维码)



第二步：等待页面加载完成后点击 Downloads，再点击右边的 python(版本号)按钮，如图所示

如果下载速度慢：

进入： repo.huaweicloud.com/python/**你刚刚看到的版本（最好减一个大版本，如 3.14.2 就输入成 3.13.2，防止 404）**

找到 python-版本号-amd64.exe（假设你的电脑是常用的 Windows）

如

1.1.2 安装 Python

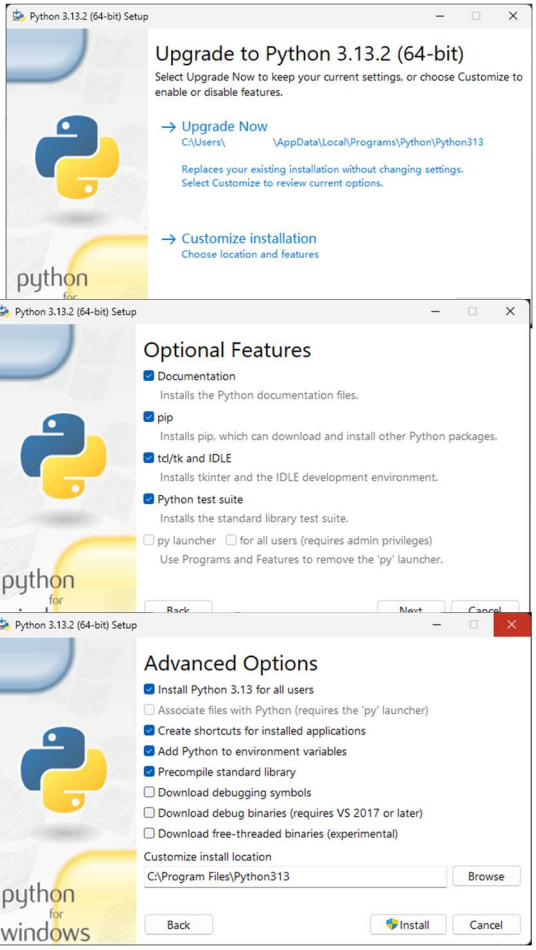
双击打开刚刚下载的安装包（如 python-3.13.2-amd64.exe）

选择下方的 **Customize installation**

在 Optional Features 界面中，直接点击 Next 按钮

按图片所示勾选后单击 Install

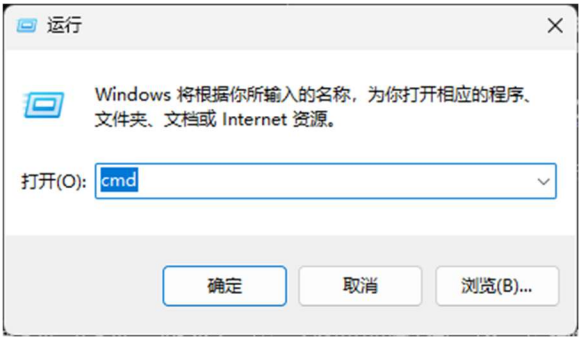
等待安装完成。



1.1.3 验证安装

在键盘上找到以下两个键

先按住左边的”Win”键，再按以下右边的”R”键会呼出运行窗口



在窗口中输入 cmd 三个字符，并点击”确定”

输入

```
python -version
```

显示了 python 的版本号就是安装成功

1.2 你的第一个 python 程序

打开开始菜单，找到 python，并打开这个程序

这是每个人的第一个 Python 编辑器，当然，更好的选择是 VSCode，安装方法和使用方法自行上网搜索，本书不再赘述。

在窗口中输入 **print**(“Hello, World”) 然后按下回车键，显示 **Hello, World** 。

后续的代码编写操作使用 VSCode，请自行安装

创建你的第一个 python 文件，在 VSCode 左侧点击”打开文件夹”，选择或创建一个文件夹用于你后续所有的代码编写。

在左侧文件夹浏览器中，按下右键点击新建文件或使用文件夹浏览器上方的图标添加文件。Python 文件的后缀名为.py，如创建”我的第一个 Python 程序.py” 这就是一个 python 文件。

双击左键打开这个 Python 文件后，在右边就可以编辑这个 Python 文件

```
print(“Hello, World”) # 输出Hello, World
```

在这段代码中，print 是 Python 中用于向控制台显示文字的函数，右边的括号代表这是一个可以被运行的函数，里面的双引号告诉 Python 这是一个直接显示的文字而不是其他东西。最右边的井号是告诉 Python 右边的不是代码的。

例如：

```
print()
```

这段代码会直接换行而不会输出任何东西，因为在括号中没有任何东西

```
print(Hello, World)
```

如果输入这个，Python 会报错，因为 Python 认为这个 Hello, World 是其他东西而不是供显示的文字。

```
print(123456)
```

直接输入这个并不会报错，因为在 Python 中，带了双引号的字符会被视为字符串，无法参与计算，但是如果是没有双引号的数字，就可以进行计算。

```
# print(“Hello, World”)
```

这段代码运行后，Python 什么也不会干，因为在井号右边的所有字符都被 Python 忽略了。

1.3 变量，数据类型，输入

1.3.1 变量

变量相当于一个盒子，在这个盒子中可以随时的放入和替换物品，同样的，变量中也可以随时**定义**和**赋值**。

定义变量是什么

定义变量 = 给盒子贴标签

如示例：

```
# 定义变量：把数字18 存入标签为"age"的盒子
age = 18

# 之后用标签就能找到数据
print(age)  # 输出：18
```

赋值变量是什么

赋值变量 = 还是同样的标签，同样的盒子，但是里面放的东西改变了。

如示例：

```
# 定义变量：把数字18 存入标签为"age"的盒子
age = 18
```

```
# 之后用标签就能找到数据
print(age)  # 输出：18

# 如果突然想改变数据
age = 19

# 那只会显示最后改变的数值
print(age)
```

变量在生活中的类似的场景也很多，举三个例子：

A：超市储物柜

- **变量名** = 柜门编号（比如"15 号柜"）
- **值** = 你存进去的包

B：微信备注

- 你好友叫"张伟"，你给他备注"二狗"
- 以后喊"二狗"就是喊他
- **二狗 = 张伟** ← 这就是变量！

C：临时便签

```
# 便利贴上写"我的零花钱"，贴到 50 元上
我的零花钱 = 50

# 买零食花了 20 元，撕下便利贴改贴到 30 元上
我的零花钱 = 30
```

在文件夹管理器中新建文件 **1-3-1.py** 并输入以下代码

```
姓名 = "小明"
年龄 = 10
成绩 = 98.5

print(姓名)
print(年龄)
print(成绩)

成绩 = 100
print("修改后的成绩：", 成绩)
```

观察代码的输出

```
小明
10
98.5
修改后的成绩： 100
```

在这个示例中我们定义了三个变量：

姓名 = "小明"

年龄 = 10

成绩 = 98.5

又使用 **print** 将它们显示在了屏幕上：

小明

10

98.5

最后再次对**成绩**进行赋值

成绩 = 100

修改后的成绩： 100

多个变量赋值

Python 允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上示例，创建一个整型对象，值为 1，从后向前赋值，三个变量被赋予相同的数值。

你也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "runoob"
```

以上示例，两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象 "runoob" 分配给变量 c。

当然，变量名也不能完全胡乱取，在 Python 中，变量名有以下几个硬性要求：

1：不能使用数字开头

2：不能与关键字（如 if，while，for 等）相同

3：不包含运算符（如+，-，*，/等）

为了以后的所有代码质量，你也应该遵守以下几个推荐要求：

1：变量名最好只使用英文，数字，下划线

2：变量名要有意义，如 a,b,c 之类的就没有意义

课后作业

在洛谷(luogu.com.cn)网站上，完成

https://www.luogu.com.cn/problem/B2001

https://www.luogu.com.cn/problem/U619327

1.3.2 数据类型

在 Python 中，数据类型是一个重要的概念，也是最容易出错的地方，基本的数据类型有：

int str float bool complex

比较复杂的内置数据类型有：

list tuple set dict

其中，Python 中的 **int float bool complex** 属于一个大类 Number（仅统称）。str 全称 String，dict 全称 Dictionary。

在 python 中，也分可变类型和不可变类型

不可变类型：Number（数字）、String（字符串）、Tuple（元组）；

可变类型：List（列表）、Dictionary（字典）、Set（集合）。

int 类型

int 类型（中文 整数类型）顾名思义，就是用于表示整数的类型，int 类型用直接输入数字表示。

```
num = 123


print(type(num))
print(type(123))
```

在这个示例中，第一行定义了一个变量 num 并赋值了 123，这个 123 是直接输入的，属于整数，观察输出结果。

```
<class 'int'>
<class 'int'>
```

输出了两个“很奇怪”的文本，实际上这是 Python 用于表示各种对象的标准格式，其中 class 的右侧正是那个数值的类型。

可以看到，这两个的数值都是'int'型，说明你已经成功输入了 int 型的数据了。

使用 int(内容)可以将任意类型转换成 int，但某些无法转换为整数的将会报错。

float 类型

float 类型（中文 浮点数），看起来似乎名字很摸不着头脑，但这个类型表示的是带有小数的量。

```
num = 1.23
num2 = 1.0


print(type(num), type(num2))
```

观察输出：

```
<class 'float'> <class 'float'>
```

可以观察到两个的结果都是浮点数 float，说明 python 看数据类型**只看形式不看结果**。

Python 内置的 float 类型使用二进制的科学计数法表示，因此有一定的误差。

```
a = 0.1
b = 0.2
print(a + b)
print(a + b == 0.3)
```

观察结果：

```
0.30000000000000004
False
```

不难发现，在 python 中，0.1+0.2 错误的等于了 0.30000000000000004，而且也不等于 0.3

使用 **float(内容)**可以将任意类型转换成 float，但某些无法转换为整数的将会报错。

bool 类型

bool 类型（中文 布尔类型），布尔类型与其他类型不同，它只有两种值：**True 和 False** 分别代表着**真和假**两种状态。

True：真，代表表达式是正确的或是一种代表开启的状态

False：与 True 相反，代表表达式错误，或是一种代表关闭的状态。

在 python 中，如果将其他类型作为 bool 进行计算，None、0 或空序列的值都会被视为 False，其余的视为 True。

complex 类型

complex 类型（中文 复数），复数在 Python 中用 a + bj 的形式表示，其中 a 是实部，b 是虚部，j 是虚数单位（数学中通常用 i，但 Python 用 j）。

```
# 创建一个复数
z = 3 + 4j

print(z)       # 输出：(3+4j)
print(type(z))  # 输出：<class 'complex'>
创建复数的几种方式：
# 方式 1：直接赋值（最常用）
c1 = 5 + 6j

# 方式 2：使用 complex()函数
c2 = complex(5, 6)      # 等同于 5 + 6j
c3 = complex(0, 2)      # 只有虚部：2j
c4 = complex(3)        # 只有实部：3 + 0j

print(c1, c2, c3, c4)   # (5+6j) (5+6j) 2j (3+0j)
```

获取复数的实部和虚部：

```
z = 7 + 8j

print("实部:", z.real)      # 输出：实部：7.0
print("虚部:", z.imag)      # 输出：虚部：8.0
```

复数的共轭：

共轭复数就是把虚部的符号变一下（+变-，-变+）。

```
z = 3 + 4j
print(z.conjugate())       # 输出：(3-4j)
```

生活中的例子：

想象你在玩地图游戏：

- 实部 = 东西方向的位置（东为正，西为负）
- 虚部 = 南北方向的位置（北为正，南为负）

- 复数 = 你在地图上的精确坐标

```
# 你的位置：向东 3 格，向北 4 格
位置 = 3 + 4j

# 朋友的相对位置：向东 1 格，向南 2 格（也就是-2j）
相对位移 = 1 - 2j

# 你们会合后的位置
会合位置 = 位置 + 相对位移
print("会合位置:", 会合位置) # 输出：会合位置：(4+2j)
```

注意事项：

1. 虚数单位必须用 `j` 或 `J`，不能用 `i`
2. 如果 `j` 前面是 `1`，不能省略：要写 `1j` 而不是 `j`
3. 实部和虚部都是浮点数类型（`float`）

```
# 错误示例
# print(3 + j)    # 报错！j 前面必须有数字

# 正确示例
print(3 + 1j)     # 输出：(3+1j)
```

类型转换：

使用 `complex()` 可以将其他类型转换为复数。

```
print(complex(5))          # 输出：(5+0j)
print(complex(5.5))        # 输出：(5.5+0j)
print(complex(True))       # 输出：(1+0j)

# 字符串也可以（但格式必须正确）
print(complex("3+4j"))     # 输出：(3+4j)
```

str 类型

str 类型（中文 字符串类型），是编程中最常用的数据类型之一，用于表示**文本**。你可以把它想象成用引号包裹起来的文字片段。

```
# 创建字符串
姓名 = "小明"
问候 = '你好'  # 单引号也可以
编号 = "001"

print(type(姓名))     # 输出: <class 'str'>
print(type("hello"))  # 输出: <class 'str'>
```

字符串的三种引号形式：

```
# 单引号
s1 = 'Hello'

# 双引号
s2 = "Python"

# 三引号（用于多行字符串）
s3 = '''这是一个
可以换行的
字符串'''

s4 = """这也是
一个多行
字符串"""

print(s3)
```

字符串拼接（合并）：

```
姓 = "张"
名 = "伟"
全名 = 姓 + 名
print("全名:", 全名)  # 输出: 全名: 张伟
```

```
# 数字和字符串不能直接相加
年龄 = 18
# print("年龄是" + 年龄)  # 这样会报错!

# 需要把数字转换成字符串
print("年龄是" + str(年龄))  # 输出：年龄是18
```

生活中的例子：

A：快递单号

- 快递单号 = "SF1234567890"
- 虽然都是数字，但它是文本，不能用来计算

B：身份证号

- 身份证 = "110101200801011234"
- 这么长的数字，如果当整数存会出问题，所以存成字符串最合适

C：游戏聊天

```
玩家A = "小火龙"
玩家B = "杰尼龟"
聊天记录 = 玩家A + "对" + 玩家B + "说：加油！"
print(聊天记录)  # 输出：小火龙对杰尼龟说：加油!
```

重要特性：

1. **字符串是不可变类型**：一旦创建，不能修改其中的某个字符

```
文字 = "hello"
# 文字[0] = "H"  # 这样会报错!
```

2. **字符串可以用加法拼接，乘法重复**

```
星星 = "*"
print(星星 * 5)  # 输出：*****
```

3. **使用 str()转换其他类型**

```
数字 = 123
字符串数字 = str(数字)
print(type(字符串数字))  # 输出：<class 'str'>
print(字符串数字)        # 输出：123（但这是文本，不是数字）
```

思考题： 为什么下面的代码会出错？怎么改正？

```
价格 = 19.9
物品 = "奶茶"
消息 = "这杯" + 物品 + "价格是" + 价格 + "元"
print(消息)
```

1.3.3 输入

在 Python 中，让用户在控制台输入文本使用 input()

运行以下代码：

```
a = input()

print(“你输入的是”, a)

b1 = input(“输入第一个数字:”)
b2 = input(“输入第二个数字:”)

print(b1 + b2)
print(int(b1 + b2))
print(int(b1) + int(b2))
```

你会发现，如果点击运行，控制台会卡住，这是因为第一行的 input 正在等待用户输入文本并回车，如在这里输入了 123 然后回车

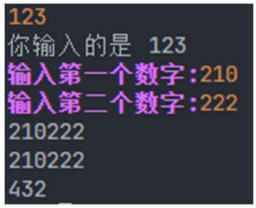
```
123
你输入的是 123
输入第一个数字:
```

在这里，你输入的 123 被正确的赋值到了变量 a 身上，并在 print 中成功调用并显示你输入的文本，然后第 5 行的 input 继续等待用户输入完成。

```
100
你输入的是 100
输入第一个数字:210
输入第二个数字:222
210222
210222
432
```

其中，加粗的字体为用户自己输入的文本。

提示：在最新版本中的 REPL，python 会自动将这些用不同颜色分出来



再次观察输出，发现到

`print(b1 + b2)` 的输出是 `210222`，也就是用户的两个输入直接拼起来的结果，这是字符串的特征。

`input` 方法返回的是一个字符串

因此，我们必须使用 `int` 将字符串转换成整数才能进行整数运算，其他类型同理。

类型转换的更多例子

除了 `int()`，其他类型也可以用类似方式转换：

```
# 转换为小数
身高 = input("请输入身高(米): ")  # 比如输入 1.75
# print(身高 * 2)          # 错误！字符串不能乘
print(float(身高) * 2)      # 正确！输出 3.5

# 转换为布尔值
is_student = input("你是学生吗？(输入 True 或 False):")
print(bool(is_student))  # 注意：非空字符串都会变成 True！
# 输入"False"也会变成 True，因为只有空字符串""才是 False
```

转换失败的情况

如果输入的内容无法转换，程序会**崩溃**：

```
年龄 = input("请输入年龄:")
age_num = int(年龄)  # 如果输入"abc"或"十八"都会报错！

运行时会看到红色错误：
ValueError: invalid literal for int() with base 10: 'abc'
```

实用技巧：去除多余空格

用户输入时经常不小心按到空格，可以用 `.strip()` 清理：

```
用户名 = input("请输入用户名:").strip()  # 自动去掉首尾空格
print("你好," + 用户名)
```

综合练习：简易计算器

在文件夹中新建 `1-3-3-calculator.py`，输入以下代码：

```
print("=== 简易计算器 ===")
num1 = input("请输入第一个数字: ")
num2 = input("请输入第二个数字: ")

# 字符串拼接
print("拼接结果: " + num1 + num2)

# 数值加法
sum_result = int(num1) + int(num2)
print("相加结果:", sum_result)

# 计算差值
diff_result = int(num1) - int(num2)
print("相减结果:", diff_result)
```

生活中的输入场景

A：超市收银系统

```
商品名 = input("扫描商品条码:")
单价 = float(input("输入单价:"))
数量 = int(input("输入数量:"))
总价 = 单价 * 数量
print("应收:", 总价, "元")
```

B：游戏角色创建

```
角色名 = input("请输入角色名称:").strip()
等级 = int(input("请输入初始等级(1-10):"))
print("欢迎", 角色名, "! 你的初始等级是", 等级)
```

课后作业

在文件夹中新建 1-3-3-homework.py，编写程序：

- 询问用户姓名、年龄、身高（米）
- 计算并输出："X 年后，姓名将年满 Y 岁"（X 由你指定，比如 5）
- 计算并输出："身高是 Y 厘米的 Z 倍"（Z 为一个小数倍数）

小贴士：REPL 交互模式

如果你在 Python 自带的 IDLE 或 VSCode 的交互窗口中，输入代码后会立即执行，非常适合测试：

```
>>> name = input("你的名字:")
你的名字:小明
>>> print("你好," + name)
你好,小明
```

而对于 .py 文件，需要点击"运行"按钮才能看到效果，两者的区别在于：**文件模式是整段运行，交互模式是逐行运行**。

1.3.2,1.3.3 课后总作业

在洛谷上 AC 以下几题：

https://www.luogu.com.cn/problem/B2003
https://www.luogu.com.cn/problem/U619333
https://www.luogu.com.cn/problem/U619353

1.4 数据处理

1.4.1 算术运算符

在 python 中，有多种算术运算符，运行以下代码：

```
# 1_4_1_1.py

print(3 + 5) # 8
print(3 - 5) # -2
print(3 * 5) # 15
print(3 / 5) # 0.6

print(7 // 5) # 1
print(7 % 5) # 2
print(2 ** 3) # 8
```

其中，//是整除运算符，会在计算除法后舍去小数部分；%是取模（取余）计算符，给出除法运算结果的余数；**是幂运算符，用于进行幂运算。

运算符优先级与括号

Python 中多个运算符一起使用时，会按优先级顺序计算。就像数学中"先乘除后加减"一样：

```
# 猜猜看结果是什么?
print(2 + 3 * 5)      # 结果是 17，不是 25！因为*先算
print((2 + 3) * 5)    # 结果是 25，括号改变了优先级

# 更复杂的例子
print(3 + 5 * 2 ** 2)      # 先算** → 再算* → 最后算+，结果是 23
print((3 + 5) * 2 ** 2)    # 结果是 32
print((3 + 5) * (2 ** 2))  # 结果也是 32
```

优先级口诀：幂运算 > 正负号 > 乘除 > 加减 > 关系运算 > not > and > or 生活中的例子：超市促销

```
原价 = 100
```

```
折扣 = 0.8
税费 = 1.1

# 错误：先加了税费再打折
错误价格 = 原价 + 原价 * 税费 * 折扣
print("错误价格:", 错误价格)  # 结果不对

# 正确：先打折，再打税
正确价格 = (原价 * 折扣) * 税费
print("正确价格:", 正确价格)  # 结果正确
```

混合类型运算规则

当 **int** 和 **float** 一起运算时，结果会变成 **float**：

```
a = 10     # int
b = 3.0    # float

print(a + b)   # 13.0 (float)
print(a - b)   # 7.0  (float)
print(a * b)   # 30.0 (float)
print(a / b)   # 3.333... (float)

# 整除和取模比较特殊
print(a // b)  # 3.0 (结果类型跟着除数走)
print(a % b)   # 1.0
```

重要提醒：字符串的"陷阱"

```
num1 = 5
num2 = "3"

# print(num1 + num2)  # 报错! int 和 str 不能直接相加
print(num1 * num2)   # 不报错! 结果是"33333", 字符串重复 5 次
print(num1 * int(num2))  # 正确! 结果是 15
```

字符串的特殊运算

虽然字符串主要不是算数类型，但 **+** 和 ***** 有特殊用途：

```
# + 是拼接
姓 = "李"
名 = "华"
全名 = 姓 + 名
print("全名:", 全名)  # 输出: 全名: 李华

# * 是重复
星号 = "★"
print(星号 * 5)      # 输出: ★★★★★
分隔线 = "-" * 20
print(分隔线)        # 输出: --------------------

# 组合应用
标题 = "  欢迎光临  "
装饰标题 = "*" + 标题 + "*"
print(装饰标题)      # 输出: *  欢迎光临  *
print("=" * len(装饰标题))  # 输出: ============
print(f"总计: {小计:.2f}元")  # :.2f 表示保留 2 位小数
print(f"实付: {应付:.2f}元")
```

1.4.2 赋值运算符

赋值运算符用于给变量赋值，最基础的就是 = 号。但 Python 还提供了更便捷的**复合赋值运算符**，能让你少写代码。

基础赋值运算符

```
# 最基本的赋值
成绩 = 95
名字 = "小红"

# 多个变量同时赋值（回顾 1.3.1）
a, b, c = 1, 2, 3
print(a, b, c)  # 输出: 1 2 3

# 相同值赋给多个变量
x = y = z = 0
print(x, y, z)  # 输出: 0 0 0
```

复合赋值运算符

这些运算符将**算术运算**和**赋值**合并为一步，让代码更简洁。

```
# 1_4_2_1.py
分数 = 100

分数 += 10   # 等价于 分数 = 分数 + 10
print(分数)  # 输出: 110

分数 -= 20   # 等价于 分数 = 分数 - 20
print(分数)  # 输出: 90

分数 *= 2    # 等价于 分数 = 分数 * 2
print(分数)  # 输出: 180

分数 //= 5   # 等价于 分数 = 分数 // 5
print(分数)  # 输出: 36
```

```
分数 %= 8    # 等价于 分数 = 分数 % 8
print(分数)  # 输出: 4

分数 **= 3   # 等价于 分数 = 分数 ** 3
print(分数)  # 输出: 64
```

常用复合赋值运算符总结表：

运算符	示例	等价于	说明
+=	a += b	a = a + b	加法赋值
-=	a -= b	a = a - b	减法赋值
*=	a *= b	a = a * b	乘法赋值
/=	a /= b	a = a / b	除法赋值
//=	a //= b	a = a // b	整除赋值
%=	a %= b	a = a % b	取模赋值
**=	a **= b	a = a ** b	幂运算赋值

生活中的例子

A：游戏积分系统

```
积分 = 0

# 击杀怪物获得100分
积分 += 100
print("当前积分:", 积分)  # 100
```

```
# 使用技能消耗 30 分
积分 -= 30
print("使用技能后:", 积分)  # 70

# 积分翻倍奖励
积分 *= 2
print("奖励后:", 积分)  # 140
```

B：零花钱管理系统

```
零花钱 = 50

# 妈妈给了 20 元
零花钱 += 20
print("收到零花钱后:", 零花钱)  # 70

# 买了 5 元的笔
零花钱 -= 5
print("买笔后:", 零花钱)  # 65

# 每周自动增加 10 元（模拟 4 周）
零花钱 += 10 * 4
print("一个月后:", 零花钱)  # 105
```

C：超市批量折扣

```
单价 = 25
数量 = 10
总价 = 0

总价 += 单价 * 数量  # 先计算再赋值
print("商品总价:", 总价)  # 250

# 满 200 打 9 折
if 总价 >= 200:
    总价 *= 0.9
```

```
print("折扣后价格:", 总价)  # 225.0
```

注意事项

```
# 错误 1：左边必须是变量
# 100 = 分数  # 会报错！不能给数字赋值

# 错误 2：变量要先定义再使用
# 计数器 += 1  # 如果之前没定义计数器，会报错！

# 正确做法
计数器 = 0
计数器 += 1
print(计数器)  # 输出：1

# 错误 3：类型不匹配
名字 = "小明"
# 名字 += 10  # 会报错！字符串不能直接加数字

# 正确做法
名字 += str(10)  # 转换成字符串
print(名字)  # 输出：小明 10
```

与算术运算符的区别

记住：**赋值运算符会改变变量的值**，而算术运算符只是计算，不改变原变量。

```
a = 10
b = 3

print(a + b)  # 输出：13
print(a)      # 输出：10（a 没变！）
```

```
a += b       # 这才是改变了a
print(a)     # 输出：13（a变了！）
```

综合练习：经验值升级系统

新建文件 1-4-2-levelup.py：

```
print("=== 游戏升级系统 ===")
经验值 = 0
等级 = 1

# 完成任务获得经验
经验值 += 150
print(f"完成任务！获得{经验值}经验")

# 击败BOSS获得经验
经验值 += 300
print(f"击败BOSS！获得{经验值}经验")

# 每500经验升1级
等级 += 经验值 // 500
经验值 %= 500  # 升级后剩余经验

print("-" * 25)
print(f"当前等级：{等级}")
print(f"当前经验：{经验值}/500")
print(f"距离下次升级还需：{500 - 经验值}经验")
```

课后作业

1. 在文件夹中新建 1-4-2-homework.py，编写程序：
   - 初始化变量 存款 = 1000
   - 模拟以下操作（直接赋值，不要用 input）：
     - 收到压岁钱 += 500

- 购买游戏机 `-= 800`
- 存款翻倍 `*= 2`
- 计算整除 100 的张数 `//= 100`，并输出"可以换 X 张百元大钞"

- 每步操作后都用 `print` 显示当前存款

2. **挑战题**：用 `+=` 实现字符串的逐次构建

```
故事 = ""
故事 += "从前有座山，"
故事 += "山里有座庙，"
故事 += "庙里有个老和尚。"
print(故事)
# 尝试用 input 让用户输入三句话，拼成一个完整故事并输出
```

小贴士：

- 复合赋值运算符虽然简洁，但初学者建议先用完整的 `a = a + b` 形式，熟练后再用 `+=`
- 写程序时多看看左边是不是变量名，避免写成 `a + b = c` 这样的错误

1.4.3 关系运算符

关系运算符（也叫比较运算符）用于比较两个值之间的关系，比较的结果是一个**布尔值**（`True` 或 `False`）。这就像生活中的"比大小"游戏。

基础关系运算符

```
# 1_4_3_1.py
a = 10
b = 20

print(a == b)  # 等于?      输出: False
print(a != b)  # 不等于?    输出: True
print(a > b)   # 大于?       输出: False
print(a < b)   # 小于?       输出: True
print(a >= b)  # 大于等于?   输出: False
print(a <= b)  # 小于等于?   输出: True
```

运算符速查表：

运算符	名称	示例	说明
==	等于	a == b	a 和 b 的值是否相等
!=	不等于	a != b	a 和 b 的值是否不相等
>	大于	a > b	a 是否大于 b
<	小于	a < b	a 是否小于 b
>=	大于等于	a >= b	a 是否大于或等于 b
<=	小于等于	a <= b	a 是否小于或等于 b

最重要：别混淆 = 和 ==

```
# 这是一个常见的错误!
年龄 = 18
# if 年龄 = 20:    # 错误! =是赋值，不是比较
#     print("你 20 岁了")  # 会报错!

# 正确写法
if 年龄 == 20:
    print("你 20 岁了")    # 这次不会进入，因为年龄是 18
```

记住： = 是给变量赋值，== 是问"相等吗？"

不同数据类型的比较规则

```
# 数字之间的比较（最常用）
print(5 > 3)       # True
print(3.14 < 3.15)  # True
```

```
print(100 == 100.0) # True（int 和 float 可以互相比较）

# 字符串比较（按字典顺序）
print("apple" < "banana")  # True，因为 a 在 b 前面
print("Z" > "a")           # False，因为大写字母在小写字母前面
print("你好" == "你好")     # True

# 字符串长度比较（不能直接比，要用 len()）
print(len("hello") > len("hi"))  # True
```

类型不同的比较会报错：

```
# print(5 > "3")  # 会报错！int 不能和 str 直接比较

# 正确做法：先转换类型
print(5 > int("3"))  # True
print(str(5) > "3")  # True（变成字符串比较）
```

生活中的例子

```
A：游戏通关判断
分数 = 8500
通关门槛 = 8000

print("是否通关:", 分数 >= 通关门槛)  # True
print("是否满分:", 分数 == 10000)     # False
```

B：年龄权限验证

```
用户年龄 = int(input("请输入你的年龄:"))

print("可以玩游戏吗:", 用户年龄 >= 12)      # 12 岁以上可以
print("需要家长陪同吗:", 用户年龄 < 8)      # 8 岁以下需要
print("是否符合优惠:", 用户年龄 <= 6)       # 6 岁以下免费
```

C：成绩评级助手

```
成绩 = float(input("请输入你的成绩:"))
```

```
print("是否及格:", 成绩 >= 60)
print("是否优秀:", 成绩 >= 90)
print("需要补考吗:", 成绩 < 60)
print("满分了吗:", 成绩 == 100)
```

综合应用：智能门禁系统

新建文件 1-4-3-security.py：

```
print("=== 小区门禁验证系统 ===")

# 预设数据
正确密码 = "123456"
管理员号码 = "888888"
用户输入密码 = input("请输入门禁密码: ")
用户号码 = input("请输入房间号: ")

# 密码验证
密码正确 = 用户输入密码 == 正确密码
号码正确 = 用户号码 == 管理员号码

print("-" * 30)
print("密码验证通过:", 密码正确)
print("管理员权限:", 号码正确)

# 可以进入的条件：密码正确 或 是管理员
可以进入 = 密码正确 or 号码正确
print("门已开启:", 可以进入)
```

字符串比较的深入理解

字符串是按 **Unicode 编码**逐个字符比较的：

```
print("abc" < "abd")     # True，因为c < d
print("100" < "20")      # True，因为"1" < "2"
print("你好" > "世界")   # False，实际编码顺序
```

```
# 如果想比较字符串表示的数字
print(int("100") < int("20"))  # False，这才是数学上的比较
```

常见错误案例

```
# 错误 1：比较浮点数的相等性
print(0.1 + 0.2 == 0.3)  # False! 回顾 1.4.1

# 正确做法
print(abs(0.1 + 0.2 - 0.3) < 0.000001)  # True

# 错误 2：忘记转换 input 类型
年龄 = input("年龄:")  # 输入 18
# print(年龄 > 18)    # 会报错！年龄是字符串"18"

# 正确做法
print(int(年龄) > 18)  # False

# 错误 3：连续比较
成绩 = 85
# if 60 <= 成绩 <= 80:  # Python 支持这种写法，但初学者容易混淆
#     print("中等")
```

课后作业

1. 在文件夹中新建 1-4-3-homework.py，完成：
   - 输入语文、数学、英语三科成绩（整数）
   - 判断是否全部及格（>=60）
   - 判断是否至少一科优秀（>=90）
   - 判断总分是否超过 250 分
   - 每判断一次都输出结果
2. **挑战题**：编写一个"石头剪刀布"判断程序

```
玩家 A = input("玩家 A 出(石头/剪刀/布):")
玩家 B = input("玩家 B 出(石头/剪刀/布):")
```

```
# 判断胜负，输出"玩家A 胜"、"玩家B 胜"或"平局"
# 提示：需要多个关系运算符组合判断
```

小贴士：

- 关系运算符的结果一定要用一个变量存起来，方便后面使用：可以进入 = 密码正确
- 调试时多用 print 输出中间结果，确认比较是否正确
- 字符串比较大小不常用，但判断是否相等（==）非常常用

1.4.4 逻辑运算符

逻辑运算符用于组合多个条件判断，就像生活中的"而且"、"或者"、"不是"。它们让程序能处理更复杂的场景。

基础逻辑运算符

Python 有三个逻辑运算符：and（与）、or（或）、not（非）。

```
# 1_4_4_1.py
a = True
b = False

print(a and b)  # 与运算：两个都为True 才为True → False
print(a or b)   # 或运算：只要一个为True 就为True → True
print(not a)    # 非运算：反过来 → False
print(not b)    # → True
```

逻辑运算符真值表：

表达式	结果	记忆口诀
`True and True`	`True`	全真才真
`True and False`	`False`	有假则假
`False and True`	`False`	有假则假
`False and False`	`False`	有假则假
`True or True`	`True`	有真则真
`True or False`	`True`	有真则真
`False or True`	`True`	有真则真
`False or False`	`False`	全假才假

生活中的例子

A：游戏权限系统

```
等级 = 15
充值金额 = 100

# 进入高级副本需要：等级≥10 级 并且 充值≥50 元
可以进入 = (等级 >= 10) and (充值金额 >= 50)
print("能进高级副本吗:", 可以进入)  # True

# 领取礼包需要：等级≥20 级 或者 是VIP
is_VIP = True
可以领取 = (等级 >= 20) or is_VIP
```

```
print("能领礼包吗:", 可以领取)  # True（虽然不是20级，但是VIP）

# 防沉迷系统：不是未成年才能无限玩
年龄 = 17
成年 = 年龄 >= 18
防沉迷限制 = not 成年
print("有防沉迷限制:", 防沉迷限制)  # True（17岁有防沉迷）
```

B：学校打卡系统

```
体温 = float(input("请输入体温:"))
有口罩 = input("是否佩戴口罩(是/否):") == "是"
健康码 = input("健康码颜色(绿/黄/红):")

# 入校条件：体温正常 且 戴口罩 且 绿码
可以入校 = (体温 <= 37.3) and 有口罩 and (健康码 == "绿")
print("允许进入学校:", 可以入校)
```

C：成绩综合评估

```
语文 = 85
数学 = 92
英语 = 78

# 三科都及格才算通过
全部及格 = (语文 >= 60) and (数学 >= 60) and (英语 >= 60)
print("是否全部及格:", 全部及格)  # True

# 至少一科优秀（≥90）就有奖励
有优秀 = (语文 >= 90) or (数学 >= 90) or (英语 >= 90)
print("是否有优秀科目:", 有优秀)  # True（数学92分）

# 不是全部优秀要加油
全部优秀 = (语文 >= 90) and (数学 >= 90) and (英语 >= 90)
需要加油 = not 全部优秀
print("需要继续努力:", 需要加油)  # True
```

短路求值特性（进阶知识）

Python 的 and 和 or 有"偷懒"特点：能确定结果后就不往后算了！

```
# and 短路：第一个为 False，后面就不执行了
print("----- and 测试 -----")
结果 = False and print("不会运行")  # 检查身份()不会被执行
print("结果:", 结果)

# or 短路：第一个为 True，后面就不执行了
print("\n----- or 测试 -----")
结果 = True or print("不会运行")  # 检查余额()不会被执行
print("结果:", 结果)
```

输出：

```
----- and 测试 -----
结果: False

----- or 测试 -----
结果: True
```

应用技巧：设置默认值

```
昵称 = input("输入昵称(直接回车用默认):") or "匿名用户"
print("你好,", 昵称)  # 如果输入为空字符串""，就显示"匿名用户"
```

运算符优先级

当多种运算符混用时，优先级很重要：not > and > or

```
结果 = True or False and not False
# 计算顺序: not False → True → False and True → False → True or False → True
print(结果)  # 输出: True

# 建议：用括号让逻辑更清晰
清晰结果 = True or (False and (not False))
print(清晰结果)  # 输出: True
```

复杂判断示例：

```
成绩 = 85
出勤率 = 0.95
违纪次数 = 0

# 评优条件：成绩优秀(≥85) 或者 (成绩良好(≥75) 且 出勤率高(≥0.9) 且 无违纪)
可以评优 = (成绩 >= 85) or ((成绩 >= 75) and (出勤率 >= 0.9) and (违纪次数 == 0))
print("可以评优:", 可以评优)  # True
```

常见错误案例

```
# 错误 1：混淆=和==
年龄 = 16
# if 年龄 = 16 or 年龄 = 18:  # 会报错! =是赋值不是比较

# 正确写法
if 年龄 == 16 or 年龄 == 18:
    print("特殊年龄")

# 错误 2：忘记括号导致逻辑混乱
成绩 = 75
# if 成绩 >= 60 and 成绩 < 70 or 成绩 >= 80:  # 容易误解

# 正确写法
if (成绩 >= 60 and 成绩 < 70) or 成绩 >= 80:
    print("符合条件")

# 错误 3：用 and/or 连接非布尔值
# Python 可以这样做，但初学者慎用
print(0 and 5)  # 输出：0（因为 0 等价于 False）
print("hello" and "world")  # 输出："world"
print("" or "默认")  # 输出："默认"
```

1.5 分支结构

1.5.1 分支语句

在上一节中，我们学习了关系运算符和逻辑运算符，它们可以判断条件是否成立。但只判断不行动就像"光说不练"。**分支结构**就是根据条件判断的结果，决定执行不同的代码路径。想象一下：如果明天下雨就带伞，否则就穿短袖——这就是生活中的分支决策。

在 Python 中，分支结构使用 **`if`** 、 **`elif`** **和** **`else`** 关键字来实现。我们先来看最简单的单分支结构。

1.5.2 if 实现单分支

单分支结构就像一条"如果...就..."的指令，**条件成立才执行特定代码，不成立就跳过。**

基本语法：

```
if 条件判断:
    # 条件为True时执行的代码（必须缩进!）
    语句块
```

关键要点：

1. `if` 后面必须是一个能得出 `True` 或 `False` 的条件表达式
2. 条件判断后面必须有**冒号** `:`
3. 执行的代码块必须**缩进**（通常是 4 个空格，按一下 Tab 键）
4. 不缩进的代码不属于 `if` 内部，不管条件是否成立都会执行

示例：游戏防沉迷系统 新建文件 `1-5-2-game.py`：

```
年龄 = int(input("请输入你的年龄: "))

if 年龄 < 18:
    print("你还未成年，游戏时间限制为 1 小时")
    print("请注意休息，保护视力")
```

```
print("游戏加载中...")  # 这行不受 if 影响，总会执行
```

运行结果分析：

- 输入 16 → 会显示防沉迷提示，然后显示"游戏加载中..."
- 输入 20 → 直接显示"游戏加载中..."

常见错误警示：

```
# 错误 1：忘记冒号
# if 年龄 < 18  # SyntaxError: expected ':'

# 错误 2：忘记缩进
# if 年龄 < 18:
# print("这行没缩进会报错！")  # IndentationError

# 错误 3：缩进不一致
# if 年龄 < 18:
#     print("第一行缩进 4 个空格")
#   print("第二行缩进 2 个空格")  # IndentationError: unexpected indent
```

综合练习：超市优惠提醒 新建文件 1-5-2-market.py：

```
总价 = float(input("请输入您的消费金额："))

if 总价 >= 200:
    折扣 = 总价 * 0.9
    print(f"恭喜！您享受 9 折优惠")
    print(f"折后价格为：{折扣:.2f}元")

print("感谢您的光临！")
```

课后小作业： 编写程序 1-5-2-homework.py：输入一个整数，如果是偶数就输出"这是个偶数"。

- **提示：**偶数的判断条件是 数字 % 2 == 0

1.5.3 if 的嵌套

嵌套分支就像"套娃"，在一个分支结构内部再套另一个分支结构。用于需要**层层判断**的场景。

语法结构：

```
if 外层条件:
    # 外层条件成立
    if 内层条件:
        # 外层和内层条件都成立
        内层代码
```

生活例子：学校体检系统 第一层：先判断是否成年 第二层：再判断体重是否正常

示例代码： 新建文件 1-5-3-nested.py：

```
年龄 = int(input("请输入年龄: "))
体重 = float(input("请输入体重(kg): "))

if 年龄 >= 18:
    print("已成年，进入成年人体检标准")

    if 体重 > 70:
        print("体重超标，建议增加运动")
    else:
        print("体重正常，请保持")
else:
    print("未成年，进入儿童体检标准")

    if 体重 > 50:
        print("青少年体重需注意")
    else:
        print("生长发育正常")
```

三级嵌套实战：成绩深度分析 新建文件 1-5-3-grade.py：

```
成绩 = float(input("请输入成绩(0-100): "))

if 成绩 >= 60:
```

```
    print("及格了！")

    if 成绩 >= 90:
        print("优秀等级")

        if 成绩 == 100:
            print("满分！你是学霸！")
        else:
            print("继续加油，向满分冲刺！")
    else:
        print("良好好等级")
else:
    print("不及格，需要努力")

    if 成绩 >= 40:
        print("还有进步空间")
    else:
        print("需要加倍努力，建议寻求帮助")
```

重要注意事项：

1. **缩进层级要清晰**：每层嵌套多缩进 4 个空格
2. **避免嵌套过深**：超过 3 层会让代码难读，可以用后面学到的逻辑运算符优化
3. **else 匹配最近 if**：else 会匹配离它最近且同一缩进层级的 if

常见错误：

```
# 错误：缩进混乱
# if 条件 1:
# print("这行没缩进")  # 报错!
#     if 条件 2:
#         print("这行缩进正确")
#        print("这行缩进不对齐")  # 报错!

# 推荐：同级别 if 对齐
# if 条件 1:
```

```
#     print("条件 1 成立")
# if 条件 2:  # 和 if 条件 1 对齐，表示独立判断
#     print("条件 2 成立")
```

课后作业： 编写程序 `1-5-3-homework.py`：输入一个数字，先判断是否为正数，如果是正数再判断是否为偶数，输出两条判断结果。

- **提示：**参考代码结构

```
数字 = int(input("输入一个整数: "))
if 数字 > 0:
    print("是正数")
    if 数字 % 2 == 0:
        print("且是偶数")
    else:
        print("但是奇数")
else:
    print("不是正数")
```

1.5.4 if 实现多分支

多分支结构就像"多选一"的选择题，用 `if-elif-else` 实现。程序会从上到下检查，**第一个满足条件的分支被执行后，后面的分支全部跳过。**

基本语法：

```
if 条件 1:
    语句块 1
elif 条件 2:  # else if 的缩写
    语句块 2
elif 条件 3:
    语句块 3
else:
    语句块 4  # 以上条件都不满足时执行
```

执行规则：

- 从上到下依次判断
- 遇到第一个 **`True`** 的条件就执行其代码块，然后**跳出整个结构**
- 如果所有条件都不满足且有 `else`，则执行 `else`
- **`elif` 可以有多个，`else` 只能有一个（可选）**

经典示例：成绩评级系统 新建文件 1-5-4-grade.py：

```
成绩 = float(input("请输入成绩(0-100): "))
if 成绩 >= 90:
    等级 = "A（优秀）"
elif 成绩 >= 80:
    等级 = "B（良好）"
elif 成绩 >= 70:
    等级 = "C（中等）"
elif 成绩 >= 60:
    等级 = "D（及格）"
else:
    等级 = "E（不及格）"

print(f"你的成绩等级是: {等级}")
```

重要理解：顺序很重要！

```
# 错误示例：顺序颠倒
成绩 = 85
if 成绩 >= 60:  # 85>=60 为 True，直接执行这里，后面的 elif 不会检查
    print("及格")
elif 成绩 >= 90:  # 这行永远不会执行
    print("优秀")  # 因为前面已经满足并跳出了

# 正确顺序：从严格到宽松
if 成绩 >= 90:
    print("优秀")
elif 成绩 >= 60:  # 这里隐含了<90 的条件
    print("及格")
else:
    print("不及格")
```

与多个独立 if 的区别：

```
# 使用 elif（多选一）
成绩 = 85
```

```
if 成绩 >= 90:
    print("优秀")
elif 成绩 >= 80:  # 前面没满足才检查这里
    print("良好")  # 输出这行
elif 成绩 >= 60:
    print("及格")  # 不会检查，因为前面已满足

# 使用多个 if（每个都判断）
成绩 = 85
if 成绩 >= 90:
    print("优秀")
if 成绩 >= 80:  # 每个 if 都独立判断
    print("良好")  # 输出这行
if 成绩 >= 60:
    print("及格")  # 也输出这行
```

数值区间判断技巧：

```
# 判断数字是否在 10 到 20 之间
数字 = 15

# 正确方法 1：使用 and
if 数字 >= 10 and 数字 <= 20:
    print("在区间内")

# 正确方法 2：Python 支持数学写法
if 10 <= 数字 <= 20:
    print("在区间内")
```

课后作业： 编写程序 1-5-4-homework.py：输入一个月份（1-12），输出对应的季节：

- 12、1、2 月 → "冬季"
- 3、4、5 月 → "春季"
- 6、7、8 月 → "夏季"
- 9、10、11 月 → "秋季"

挑战题： 编写程序 1-5-4-bmi.py：输入身高（米）和体重（kg），计算 BMI 值，并按以下标准输出健康状态：

- BMI < 18.5 → "偏瘦"
- 18.5 ≤ BMI < 24 → "正常"
- 24 ≤ BMI < 28 → "超重"
- BMI ≥ 28 → "肥胖"

计算公式：BMI = 体重 / (身高 ** 2)

1.5.5 分支结构实践

综合案例：智能地铁购票系统 新建文件 1-5-5-subway.py：

```
print("=== 欢迎使用智能地铁购票系统 ===")
print("=== 支持 1 号线、2 号线、10 号线 ===")

# 输入阶段
线路 = int(input("请输入线路号(1/2/10): "))
起点 = input("请输入起点站: ")
终点 = input("请输入终点站: ")
距离 = float(input("请输入距离(km): "))

# 初始化票价
票价 = 0

# 不同线路不同计价规则
if 线路 == 1:
    # 1 号线：起步 2 元(6km 内)，超出每 km 加 0.5 元，最高 8 元
    if 距离 <= 6:
        票价 = 2
    elif 距离 <= 12:
        票价 = 2 + (距离 - 6) * 0.5
    else:
        票价 = min(5 + (距离 - 12) * 0.3, 8)  # 最高 8 元

    print(f"1 号线票价: {票价:.1f}元")

elif 线路 == 2:
    # 2 号线：统一票价 3 元
```

```
    票价 = 3
    print(f"2 号线票价: {票价}元")

elif 线路 == 10:
    # 10 号线: 高峰期(7-9 点,17-19 点)加价 50%
    时间 = int(input("请输入当前小时(0-23): "))

    if 7 <= 时间 <= 9 or 17 <= 时间 <= 19:
        基础票价 = 距离 * 0.6
        票价 = 基础票价 * 1.5
        print(f"10 号线高峰期票价: {票价:.1f}元")
    else:
        票价 = 距离 * 0.6
        print(f"10 号线平峰期票价: {票价:.1f}元")

else:
    print("抱歉，暂不支持该线路")

# 优惠判断（学生半价）
是否是学生 = input("是否持有学生证(是/否): ") == "是"
if 是否是学生:
    票价 = 票价 * 0.5
    print(f"学生优惠后票价: {票价:.1f}元")

print("-" * 30)
print(f"最终应付: {票价:.1f}元")
print("请在闸机处扫码进站")
```

课后总作业（洛谷专项）： 在洛谷上 AC 以下题目，这些题目需要综合运用本章的分支结构知识：
https://www.luogu.com.cn/problem/U619357
https://www.luogu.com.cn/training/101 （这是题单，有多个题目）

挑战综合项目： 编写程序 `1-5-5-project.py`：实现一个简单的"登录-菜单"系统

1. 预设用户名和密码（如"admin"和"123456"）
2. 输入用户名和密码
3. 验证是否匹配，匹配则显示"登录成功"并进入菜单，否则提示"用户名或密码错误"
4. 登录成功后显示菜单：
    - 按 1：显示当前时间（可用简单文本代替）
    - 按 2：进入计算器功能
    - 按 3：退出系统
5. 使用嵌套 `if` 和多分支结构实现

小贴士：

- **调试技巧**：在复杂分支中多用 `print()` 输出中间结果，确认条件判断是否正确
- **代码可读性**：当分支嵌套超过 3 层时，考虑用逻辑运算符 `and/or` 简化，或拆分成函数
- **测试要全面**：每个分支、每个条件都要测试到，特别是边界值（如成绩刚好 60 分、BMI 刚好 24）
- **优先级记忆口诀**：`not > and > or >` 关系运算符 `>` 算术运算符，不确定就加括号 `()`

本章节小结： 通过 1.5 节的学习，你掌握了程序"做决策"的核心能力：

- 使用 `if` 实现单分支判断
- 使用 `elif/else` 实现多选一
- 合理嵌套分支处理复杂逻辑
- 避开常见语法陷阱（冒号、缩进、=与==）

记住：**好的分支结构像一棵清晰的决策树**，让程序能根据不同情况做出智能反应。这是解决实际问题的关键能力！

1.6 循环结构

1.6.1 为什么需要循环

想象你要打印 10 句"我爱学 Python"，如果逐行写 10 次 print()，代码会变得又长又笨。生活中也充满重复：每天做眼保健操、体育课跑步 10 圈、游戏刷怪升级……**循环结构就是让程序自动重复执行代码的利器，用少量代码解决重复性工作。**

Python 有两种主要循环：

- **while** 循环：条件满足就持续执行（不确定次数）
- **for** 循环：遍历序列逐次执行（确定次数）

1.6.2 while 循环

基本语法：

```
while 循环条件:
    # 条件为True 时重复执行的代码块
    循环体
```

执行流程：

1. 判断循环条件是否为 True
2. 如果是，执行循环体所有代码
3. 执行完回到第 1 步再次判断
4. 如果条件为 False，跳出循环，执行后面的代码

核心要点： 循环体内**必须**有改变条件的语句，否则会变成**死循环**（程序卡死）！

示例：倒计时器 新建文件 1-6-2-countdown.py：

```
print("=== 火箭发射倒计时 ===")
倒计时 = 10

while 倒计时 > 0:
    print(f"{倒计时}...")
    倒计时 -= 1  # 关键：每次减1，最终会让条件变False
    # 这里的缩进代码会重复执行10 次
```

```
print("点火！")
```

运行结果：

```
10...
9...
8...
...
1...
点火！
```

常见错误：死循环

```
# 错误示例：忘记改变条件
# 倒计时 = 10
# while 倒计时 > 0:
#     print("永远卡在这里！")
#     # 没有倒计时 -= 1，条件永远为 True

# 如何强制停止死循环？
# 在终端按 Ctrl+C 终止程序
```

综合练习：累加求和 新建文件 1-6-2-sum.py：

```
print("=== 计算 1+2+3+...+100 的和 ===")
数字 = 1
总和 = 0

while 数字 <= 100:
    总和 += 数字  # 等价于 总和 = 总和 + 数字
    print(f"当前数字：{数字}，累计和：{总和}")  # 调试用
    数字 += 1

print(f"最终总和：{总和}")
```

生活中的例子：

- **存钱计划**：每天存 10 元，直到存满 1000 元买游戏机
- **背单词**：每天背 20 个单词，连续坚持 30 天
- **游戏刷怪**：血量>0 就一直攻击，血量为 0 退出战斗

课后作业： 编写程序 1-6-2-homework.py：输入一个正整数 n，计算 n 的阶乘（n! = 1×2×3×...×n）

- **提示**：结果可能很大，建议从 n=5 或 n=10 开始测试
- **优化挑战**：加入判断，如果 n>20 提示"计算量过大"

1.6.3 for 循环与 range()

for 循环更适合**已知次数**的重复，配合 range()函数威力强大！

基本语法：

```
for 循环变量 in range(次数):
    # 重复执行的代码
```

range()函数详解：

- range(5) → 生成 0, 1, 2, 3, 4（共 5 个数，从 0 开始）
- range(1, 6) → 生成 1, 2, 3, 4, 5（从 1 到 5，不包含 6）
- range(1, 11, 2) → 生成 1, 3, 5, 7, 9（步长为 2）

示例：打印乘法表 新建文件 1-6-3-multiplication.py：

```
print("=== 打印 5 的乘法表 ===")

for i in range(1, 10):  # i 会依次等于 1,2,3,...,9
    print(f"5 × {i} = {5 * i}")

print("\n=== 倒序打印 ===")
for i in range(9, 0, -1):  # 步长为-1，从 9 到 1
    print(f"5 × {i} = {5 * i}")
```

对比 while 和 for：

```
# while 版本：打印 1-10
i = 1
while i <= 10:
    print(i)
    i += 1
```

```
# for 版本：更简洁
for i in range(1, 11):
    print(i)
```

综合练习：统计偶数个数 新建文件 1-6-3-even.py：

```
print("=== 统计 1-100 中的偶数 ===")
偶数个数 = 0

for 数字 in range(1, 101):
    if 数字 % 2 == 0:  # 分支结构嵌套在循环内!
        偶数个数 += 1
        print(f"发现偶数: {数字}")

print(f"1-100 中共有 {偶数个数} 个偶数")
```

课后作业：

1. 编写程序 1-6-3-homework1.py：用 for 循环计算 1 到 100 的奇数和
2. 编写程序 1-6-3-homework2.py：输入 n，输出 n 行 "*"组成的等腰三角形（挑战）

```
n=3 时：
  *
 ***
*****
```

1.6.4 break 和 continue

break：立即结束整个循环，跳出循环体 continue：跳过本次循环，直接进入下一次

---

break 示例：查找第一个能被 7 整除的数 新建文件 1-6-4-break.py：

```
print("=== 查找 1-100 中第一个能被 7 整除的数 ===")

for 数字 in range(1, 101):

    if 数字 % 7 == 0:
        print(f"找到了！是 {数字}")
        break  # 找到了就立即退出，不再检查后面的数
    print(f"检查 {数字} 不符合")

print("循环结束")
```

continue 示例：跳过 3 的倍数 新建文件 1-6-4-continue.py：

```
print("=== 打印 1-10 中非 3 的倍数 ===")

for 数字 in range(1, 11):
    if 数字 % 3 == 0:
        continue  # 是 3 的倍数就跳过，不执行后面的 print
    print(数字)  # 这行在 continue 之后，被跳过时不会执行

print("循环结束")
```

对比表格：

关键字	作用	生活比喻
break	彻底退出循环	跑步时受伤，直接退出比赛
continue	跳过当前轮次	跑步时鞋带松了，系好继续跑下一圈

常见错误：

```
# 错误：break 缩进不对
for i in range(5):
    if i == 3:
    break  # IndentationError: break 要缩进

# 正确
for i in range(5):
    if i == 3:
        break
```

课后作业： 编写程序 1-6-4-homework.py：让用户不停输入数字，直到输入"0"时结束，并计算所有输入数字的总和（用 break 实现）

1.6.5 循环嵌套

循环嵌套是一个循环内部再套一个循环，常用于处理二维结构。

执行顺序： 外层循环执行 1 次，内层循环要执行完整的一轮

示例：打印九九乘法表 新建文件 `1-6-5-nested.py`：

```
print("=== 九九乘法表 ===")

for i in range(1, 10):  # 外层：控制行数（1-9 行）
    for j in range(1, i + 1):  # 内层：每行打印 i 个式子
        print(f"{j}×{i}={i*j}", end="  ")  # end=" "表示不换行，用空格分隔
    print()  # 内层循环结束后换行

print("\n=== 用循环画矩形 ===")
高 = int(input("输入高度: "))
宽 = int(input("输入宽度: "))

for i in range(高):
    for j in range(宽):
        print("*", end="")
    print()  # 换行
```

运行结果：

```
=== 九九乘法表 ===
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
```

```
=== 用循环画矩形 ===
输入高度: 4
输入宽度: 2
**
**
**
```

**生活例子：

- **工作日和时间段**：外层循环 5 个工作日，内层循环 8 个上课时段
- **电影院座位**：外层循环排数（A-J），内层循环每排的座位号（1-10）

综合练习：统计素数 新建文件 1-6-5-prime.py：

```
print("=== 统计 1-100 中的素数 ===")
素数个数 = 0

for 数字 in range(2, 101):  # 从 2 开始，1 不是素数
    是素数 = True

    # 内层循环：检查是否能被 2 到数字-1 之间的数整除
    for 除数 in range(2, 数字):
        if 数字 % 除数 == 0:
            是素数 = False
            break  # 发现能整除就不是素数，立即跳出内层循环

    if 是素数:
        print(f"{数字} 是素数")
        素数个数 += 1

print(f"1-100 中共有 {素数个数} 个素数")
```

课后作业： 编写程序 1-6-5-homework.py：用循环嵌套打印以下图案（n=5）

```
*****
****
***
**
*
```

•**提示**：外层控制行，内层控制每行的*数量，数量与行号有关

1.6.6 循环综合实践

综合案例：简易 ATM 系统 新建文件 1-6-6-atm.py：

```
print("=== 简易 ATM 系统 ===")
密码 = "123456"
余额 = 1000

# 登录验证（最多3 次）
for 尝试次数 in range(3):
    输入密码 = input("请输入 6 位密码: ")
    if 输入密码 == 密码:
        print("登录成功! ")
        break
    else:
        print(f"密码错误! 还剩 {2 - 尝试次数} 次机会")
else:
    # for 循环正常结束（没遇到break）才执行
    print("三次错误，卡已锁定! ")
    exit()  # 退出程序

# 主菜单循环
while True:
    print("\n" + "="*20)
    print("1. 查询余额")
    print("2. 存款")
```

```
    print("3. 取款")
    print("4. 退出")

    选择 = input("请选择操作(1-4): ")

    if 选择 == "1":
        print(f"当前余额: {余额}元")

    elif 选择 == "2":
        金额 = float(input("请输入存款金额: "))
        if 金额 > 0:
            余额 += 金额
            print(f"存款成功! 存入{金额}元")
        else:
            print("金额必须为正数! ")

    elif 选择 == "3":
        金额 = float(input("请输入取款金额: "))
        if 金额 <= 余额 and 金额 > 0:
            余额 -= 金额
            print(f"取款成功! 取出{金额}元")
        else:
            print("余额不足或金额无效! ")

    elif 选择 == "4":
        print("感谢使用，再见! ")
        break  # 退出主循环

    else:
        print("无效选择，请重新输入! ")
```

洛谷专项训练： 循环结构是编程的核心，务必在洛谷上完成以下题目巩固：

https://www.luogu.com.cn/problem/U619360
https://www.luogu.com.cn/problem/U619368

本章节小结： 通过 1.6 节，你掌握了让程序"重复执行"的魔法：

- while 循环：适合不确定次数的条件循环
- for+range()：适合确定次数的计数循环
- break/continue：精准控制循环流程
- 循环嵌套：处理二维复杂问题

重要提醒：

1. **死循环是初学者最大敌人**，务必确保循环条件会变为 False
2. **缩进层级要清晰**，循环体内缩进，循环体外不缩进
3. **先写简单循环，再逐步添加分支和嵌套**，不要一次性写太复杂

循环结构+分支结构，你已经掌握了编程的两大核心逻辑。接下来就能写出解决实际问题的完整程序了！

1.7 列表

1.7.1 什么是列表

列表（List）是 Python 中最常用的数据结构之一，就像一个可以自由伸缩的"超级收纳盒"，能按顺序存放多个数据，还能随时增删改查。想象一下你的购物清单、班级名单、游戏道具栏—这些都是现实生活中的"列表"。

创建列表：

```
# 创建一个空列表（就像拿了个空盒子）
shopping_list = []

# 创建带有初始值的列表
student_names = ["小明", "小红", "小刚", "小丽"]
scores = [95, 88, 92, 78, 65]
mixed_list = ["hello", 123, 3.14, True]  # 列表可以混合不同类型的数据
```

生活中的例子：

- **超市购物清单**：["鸡蛋", "牛奶", "面包", "苹果"]
- **游戏背包**：["铁剑", "生命药水", "魔法卷轴", "金币"]
- **一周课程表**：["语文", "数学", "英语", "物理", "化学"]

核心特点：

1. **有序性**：每个元素都有固定位置（索引）
2. **可变性**：可以随时修改、添加、删除元素
3. **异构性**：可以储存不同类型数据
4. **动态性**：长度可以随时变化

1.7.2 列表的索引与访问

列表中的每个元素都有一个编号，叫做**索引（index）**。Python 的索引从 0 开始（这是编程世界的普遍规则）。

```
# 1_7_2_index.py
subjects = ["语文", "数学", "英语", "物理", "化学"]
```

```
# 正向索引（从 0 开始）
print(subjects[0])  # 输出：语文
print(subjects[1])  # 输出：数学
print(subjects[4])  # 输出：化学

# 反向索引（从-1 开始，-1 代表最后一个）
print(subjects[-1])  # 输出：化学
print(subjects[-2])  # 输出：物理
```

常见错误警示：

```
# 错误 1：索引越界
# print(subjects[5])  # IndexError: list index out of range

# 错误 2：空列表索引
empty_list = []
# print(empty_list[0])  # IndexError: list index out of range

# 正确做法：先检查长度
if len(subjects) > 5:
    print(subjects[5])
else:
    print("索引超出范围")
```

获取列表长度：

```
print(len(subjects))  # 输出：5
```

1.7.3 列表的切片操作

切片（Slicing）就像从长面包上切下一段，可以一次性获取列表中的连续多个元素。

```
# 1_7_3_slice.py
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 基本语法：list[start:end:step]（包含 start，不包含 end）
```

```
print(numbers[2:5])      # 输出: [2, 3, 4]（索引 2 到 4）
print(numbers[:4])       # 输出: [0, 1, 2, 3]（省略 start，默认从 0 开始）
print(numbers[5:])       # 输出: [5, 6, 7, 8, 9]（省略 end，默认到最后）
print(numbers[:])        # 输出: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]（复制整个列表）

# 带步长的切片
print(numbers[1:8:2])    # 输出: [1, 3, 5, 7]（从 1 到 7，步长 2）
print(numbers[::-1])     # 输出: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]（步长-1，倒序）
```

生活中的例子：

- **班级前 5 名**：`scores[:5]`（省略 `start`，取前 5 个）
- **倒数 3 名**：`scores[-3:]`（从倒数第 3 个到最后）
- **每隔一个取一个**：`students[::2]`（步长为 2）

1.7.4 修改列表元素

列表是可变类型，可以直接通过索引修改元素。

```
# 1_7_4_modify.py
fruits = ["apple", "banana", "orange", "grape"]

# 修改单个元素
fruits[1] = "pear"  # 把索引 1 的"banana"改成"pear"
print(fruits)  # 输出: ['apple', 'pear', 'orange', 'grape']

# 通过切片批量修改（会覆盖原内容）
fruits[1:3] = ["watermelon", "strawberry"]
print(fruits)  # 输出: ['apple', 'watermelon', 'strawberry', 'grape']

# 切片替换数量可以不等
fruits[1:2] = ["a", "b", "c"]  # 1 个元素被替换成 3 个
print(fruits)  # 输出: ['apple', 'a', 'b', 'c', 'strawberry', 'grape']
```

常见错误：

```
# 错误：字符串是不可变类型，不能这样修改
name = "zhangsan"
# name[0] = "Z"  # TypeError: 'str' object does not support item assignment

# 正确：列表可以修改
name_list = list(name)  # 字符串转列表：['z', 'h', 'a', 'n', 'g', 's', 'a',
'n']
name_list[0] = "Z"
new_name = "".join(name_list)  # 列表转字符串
print(new_name)  # 输出：Zhangsan
```

1.7.5 列表常用方法（增删操作）

Python 为列表提供了丰富的内置方法，就像收纳盒的各种功能按钮。

```
# 1_7_5_methods.py
tools = ["hammer", "screwdriver"]

# 1. append() - 在末尾添加一个元素（就像往箱子后面塞东西）
tools.append("wrench")
print(tools)  # 输出：['hammer', 'screwdriver', 'wrench']

# 2. extend() - 在末尾添加多个元素（合并另一个列表）
tools.extend(["pliers", "saw"])
print(tools)  # 输出：['hammer', 'screwdriver', 'wrench', 'pliers', 'saw']

# 3. insert() - 在指定位置插入元素
tools.insert(1, "drill")  # 在索引 1 位置插入
print(tools)  # 输出：['hammer', 'drill', 'screwdriver', 'wrench', 'pliers',
'saw']

# 4. remove() - 删除第一个匹配的元素（按值删除）
tools.remove("wrench")
print(tools)  # 输出：['hammer', 'drill', 'screwdriver', 'pliers', 'saw']
```

```
# 5. pop() - 删除并返回指定位置的元素（按索引删除，默认最后一个）
last_tool = tools.pop()  # 删除最后一个元素
print(f"被移除的工具: {last_tool}")  # 输出: 被移除的工具: saw
print(tools)  # 输出: ['hammer', 'drill', 'screwdriver', 'pliers']

# 6. clear() - 清空整个列表
tools.clear()
print(tools)  # 输出: []
```

1.7.6 列表的其他实用操作

```
# 1_7_6_other.py
nums = [5, 2, 8, 1, 9, 3]

# 1. count() - 统计元素出现次数
print(nums.count(2))  # 输出: 1

# 2. index() - 查找元素第一次出现的位置
print(nums.index(8))  # 输出: 2

# 3. reverse() - 原地反转列表
nums.reverse()
print(nums)  # 输出: [3, 9, 1, 8, 2, 5]

# 4. sort() - 原地排序（默认升序）
nums.sort()
print(nums)  # 输出: [1, 2, 3, 5, 8, 9]

nums.sort(reverse=True)  # 降序
print(nums)  # 输出: [9, 8, 5, 3, 2, 1]

# 5. 复制列表（重要!）
original = [1, 2, 3]
```

```
copy1 = original  # 这是引用赋值，两个变量指向同一个列表
copy1[0] = 999
print(original)  # 输出: [999, 2, 3]（原列表被修改了!）

# 正确复制方法
original = [1, 2, 3]
copy2 = original.copy()  # 或者 copy2 = original[:]
copy2[0] = 999
print(original)  # 输出: [1, 2, 3]（原列表不变）
print(copy2)     # 输出: [999, 2, 3]
```

1.7.7 *列表推导式（选学）

列表推导式是 Python 的优雅特性，可以用一行代码创建列表。

```
# 1_7_7_comprehension.py
# 传统方法：生成 1-10 的平方列表
squares = []
for i in range(1, 11):
    squares.append(i ** 2)
print(squares)  # 输出: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# 列表推导式：更简洁
squares = [i ** 2 for i in range(1, 11)]
print(squares)  # 输出: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# 带条件筛选：生成偶数的平方
even_squares = [i ** 2 for i in range(1, 11) if i % 2 == 0]
print(even_squares)  # 输出: [4, 16, 36, 64, 100]
```

1.7.8 循环与列表的综合应用

遍历列表：

```
# 1_7_8_loop.py
students = ["Alice", "Bob", "Charlie", "David"]

# 方式 1：直接遍历元素
for student in students:
    print(f"你好，{student}")

# 方式 2：遍历索引（需要修改元素时）
for i in range(len(students)):
    students[i] = "优秀-" + students[i]
print(students)  # 输出：['优秀-Alice', '优秀-Bob', '优秀-Charlie', '优秀-
David']
```

综合案例：成绩统计分析

```
# 1_7_8_grades.py
scores = [85, 92, 78, 65, 88, 95, 72, 90]

# 计算总分和平均分
total = sum(scores)  # Python 内置函数
average = total / len(scores)
print(f"总分：{total}，平均分：{average:.2f}")

# 统计优秀人数(>=90)
excellent_count = 0
for score in scores:
    if score >= 90:
        excellent_count += 1
print(f"优秀人数：{excellent_count}")

# 找出最高分
max_score = scores[0]
for score in scores:
```

```
    if score > max_score:
        max_score = score
print(f"最高分: {max_score}")
```

课后作业

基础作业：

1. 在文件夹中新建 1-7-homework1.py：
   - 创建一个列表 my_friends = ["Tom", "Jerry", "Spike", "Tyke"]
   - 打印第 2 个和第 4 个朋友
   - 将第 3 个朋友改为"NewFriend"
   - 在末尾添加"LastFriend"
   - 删除第一个朋友
   - 打印最终列表长度
2. 在文件夹中新建 1-7-homework2.py：
   - 创建列表 numbers = [23, 45, 12, 67, 89, 34]
   - 使用循环找出其中的最大值和最小值
   - 计算所有元素的平均值
   - 统计大于 50 的数有几个

挑战作业： 3. 在文件夹中新建 1-7-homework3.py：

- 创建两个列表 list_a = [1, 2, 3] 和 list_b = [4, 5, 6]
- 合并成一个新列表 list_c
- 将 list_c 反转
- 排序后输出

洛谷专项训练： 完成以下需要列表思维的题目：
https://www.luogu.com.cn/problem/U628035

本章节小结

通过 1.7 节的学习，你掌握了 Python 中最重要的数据结构之一：

- **创建列表**：用[]创建，可包含任意类型元素
- **访问元素**：通过索引 list[i]，支持正向和反向
- **切片操作**：list[start:end:step]获取子列表

- **修改列表**：直接赋值修改，用 append/insert 添加，remove/pop 删除
- **实用方法**：len/count/index/sort/reverse/copy 等
- **循环遍历**：for item in list 是常用模式
- **列表推导式**：优雅地创建列表（进阶）

重要提醒：

1. **索引从 0 开始**：这是最容易犯错的地方，第 1 个元素是 list[0]
2. **区分方法与函数**：len(list)是函数，list.append()是方法
3. **复制列表**：必须用 copy()或切片[:]，否则只是引用
4. **列表是可变的**：这是列表与字符串、元组的核心区别

列表配合循环和分支结构，你已经具备了处理批量数据的能力。下一章我们将学习更高级的数据结构——字典！

小贴士： 调试列表时，多用 print()输出中间状态，确认元素是否正确添加或删除。列表操作是后续所有复杂程序的基础，务必通过大量练习熟练掌握！

2 语法进阶

2.1 函数基础

2.1.1 为什么需要函数

想象你要做一个"欢迎界面"，需要打印 10 行星号，然后打印标题，再打印 10 行星号。如果不用函数，你得把 print("*" * 20)重复写 20 遍！这就像每次吃饭都要重新发明碗筷一样荒唐。

函数就是给一段代码起个名字，以后直接喊名字就能用。它解决了三大痛点：

1. **代码复用**：写一次，用无数次
2. **模块化**：把大程序拆成小块，像拼乐高一样管理
3. **可读性**：计算 BMI()比一堆公式清晰 100 倍

生活中的例子：

- **外卖点餐**：你不用自己买菜→洗菜→炒菜，直接调用"点外卖()"函数
- **洗衣机**：把"加水→旋转→排水→脱水"封装成"洗衣服()"函数
- **数学公式**：f(x)=2x+1 就是最早的函数概念，输入 3 输出 7

2.1.2 函数的定义与调用

定义函数：给代码块贴标签

```
# 2_1_2_define.py
def 打招呼():
    """这个函数会打印三行欢迎信息"""  # 文档字符串，说明函数作用
    print("*" * 20)
    print("欢迎来到 Python 世界！")
    print("*" * 20)
```

语法要点：

- def 是定义函数的关键词（define 的缩写）
- 打招呼 是函数名，要符合变量命名规则（见 1.3.1 节）
- () 里暂时为空，表示不需要输入参数

- **：**冒号不能少，表示接下来是函数体
- **函数体必须缩进 4 个空格**（和 `if/for` 一样）
- `"""`文档字符串`"""` 是好习惯，说明函数是干嘛的

调用函数：喊名字执行

```
# 调用刚才定义的函数
打招呼()  # 输出三行星号欢迎语

# 想欢迎几次就调用几次
for i in range(3):
    打招呼()
```

执行流程：

1. 遇到打招呼()时，程序跳转到函数定义处
2. 按顺序执行函数体内的代码
3. 执行完回到调用处，继续往下走

常见错误警示：

```
# 错误 1：定义时多缩进
#   def 打招呼():  # IndentationError: unexpected indent
#       print("...")

# 错误 2：调用在定义之前
# 打招呼()  # NameError: name '打招呼' is not defined
# def 打招呼():
#     print("...")

# 错误 3：忘记括号
# 打招呼   # 不会执行！只是引用了函数对象本身
```

2.1.3 函数的参数：让函数更灵活

刚才的函数只能打印固定内容，**参数让函数接收外部输入**，就像数学函数 f(x)里的 x。

位置参数：按顺序传值

```
# 2_1_3_param.py
def 个性化打招呼(名字, 次数):
    """根据名字和次数打印个性化欢迎"""
    for i in range(次数):
        print(f"{i+1}. 欢迎 {名字} 同学! ")

# 调用时按顺序传参
个性化打招呼("小明", 3)
# 输出:
# 1. 欢迎 小明 同学!
# 2. 欢迎 小明 同学!
# 3. 欢迎 小明 同学!

# 参数可变数
个性化打招呼("小红", 2)
个性化打招呼("全班", 5)
```

生活中的例子：

- **外卖函数**：点外卖(菜品，数量，地址)
- **洗衣机函数**：洗衣服(衣服，模式，时间)
- **游戏函数**：发射火球(方向，威力)

默认参数：给参数默认值

有些参数经常不变，可以设默认值，调用时可省略。

```
def 订奶茶(口味, 甜度="半糖", 冰度="少冰"):
    """默认半糖少冰，可指定"""
    print(f"订购{甜度}{冰度的{口味}奶茶")

订奶茶("珍珠奶茶")  # 用默认的半糖少冰
订奶茶("芋泥波波", "全糖")  # 只改甜度
订奶茶("椰果", "无糖", "去冰")  # 全改
```

规则：

- 默认参数必须放在**位置参数后面**
- `def 订奶茶(口味, 甜度="半糖")` ✓
- `def 订奶茶(甜度="半糖", 口味)` × **SyntaxError**

2.1.4 函数的返回值：把结果送回来

函数不仅能执行，还能**把计算结果返回给调用者**，就像数学函数算出答案一样。

```
# 2_1_4_return.py
def 计算圆面积(半径):
    """返回圆的面积"""
    面积 = 3.14159 * 半径  ** 2
    return 面积  # 把结果送回去

# 调用并接收返回值
r = 5
result = 计算圆面积(r)  # result 接收返回的 78.53975
print(f"半径{r}的圆面积是{result:.2f}")

# 返回值可以参与运算
总面积 = 计算圆面积(3) + 计算圆面积(4)
print(f"两个圆总面积: {总面积:.2f}")
```

return 的魔法：

立即结束函数 ：return 后函数立刻退出

```
def 判断奇偶(数):
    if 数 % 2 == 0:
        return "偶数"  # 执行到这里就返回，后面不执行
    return "奇数"  # else 可以省略
```

返回多个值：用逗号分隔，实际是返回元组

```
def 计算矩形(长, 宽):
    面积 = 长 * 宽
    周长 = 2 * (长 + 宽)
    return 面积, 周长  # 返回两个值
```

```
a, c = 计算矩形(3, 4)  # 解包赋值
print(f"面积{a}, 周长{c}")  # 输出: 面积12, 周长14
```

常见错误警示：

```
# 错误1: 没有return却想用返回值
def 打招呼():
    print("你好")

消息 = 打招呼()  # 消息是None!
print(消息)  # 输出: None

# 错误2: return后面没值
def 空返回():
    return  # 返回None

# 错误3: 忘记接收返回值
def 加一(x):
    return x + 1

加一(5)  # 返回6但被丢弃了，什么也没发生
print(加一(5))  # 正确! 打印6
```

2.1.5 变量的作用域：你的变量在哪有效？

```
# 2_1_5_scope.py
全局变量 = 100  # 在函数外部定义

def 测试作用域():
    局部变量 = 200  # 在函数内部定义
    print(f"函数内看局部变量：{局部变量}")  # 可以访问
    print(f"函数内看全局变量：{全局变量}")  # 可以读取

测试作用域()
# print(局部变量)  # NameError：函数外无法访问局部变量
print(f"函数外看全局变量：{全局变量}")
```

作用域规则

- **局部变量**：函数内部定义，只在函数内有效
- **全局变量**：函数外部定义，整个文件有效
- **函数优先**：函数内如果和全局同名，优先用局部的

在函数内修改全局变量：

```
计数器 = 0

def 增加计数():
    global 计数器  # 声明要修改全局变量
    计数器 += 1    # 否则会被当作创建局部变量

增加计数()
print(计数器)  # 输出：1
```

常见错误警示：

```
x = 10

def 错误示范():
    x = x + 1  # UnboundLocalError：先赋值 x 被视为局部变量，但右边 x 未定义

# 正确做法
def 正确示范():
```

```
    global x
    x = x + 1
```

2.1.6 函数综合实践

案例：智能购物小票系统

```
# 2_1_6_shopping.py
def 计算折扣价(原价, 会员等级):
    """根据会员等级计算折扣"""
    if 会员等级 == "VIP":
        return 原价 * 0.7
    elif 会员等级 == "黄金":
        return 原价 * 0.8
    elif 会员等级 == "白银":
        return 原价 * 0.9
    else:
        return 原价

def 打印小票(商品列表, 总价, 实付):
    """格式化打印购物小票"""
    print("=" * 30)
    print("       智能购物小票")
    print("=" * 30)
    for 商品, 单价 in 商品列表:
        print(f"{商品:<10} ¥{单价:>8.2f}")
    print("-" * 30)
    print(f"{'总价:':<10} ¥{总价:>8.2f}")
    print(f"{'实付:':<10} ¥{实付:>8.2f}")
    print(f"{'节省:':<10} ¥{总价-实付:>8.2f}")
    print("=" * 30)

# 主程序
商品 = [("苹果", 12.5), ("牛奶", 35.0), ("面包", 8.8)]
总价 = sum([价格 for _, 价格 in 商品])  # 列表推导求和
```

```
等级 = input("请输入会员等级(VIP/黄金/白银/普通): ")

实付 = 计算折扣价(总价, 等级)
打印小票(商品, 总价, 实付)
```

运行效果：

```
请输入会员等级(VIP/黄金/白银/普通): VIP
==============================
       智能购物小票
==============================
苹果          ¥  12.50
牛奶          ¥  35.00
面包          ¥   8.80
------------------------------
总价:         ¥  56.30
实付:         ¥  39.41
节省:         ¥  16.89
==============================
```

课后作业

基础作业：

1. 新建 2-1-homework1.py：编写函数计算 BMI(身高, 体重)，返回 BMI 值和状态（偏瘦/正常/超重/肥胖）
2. 新建 2-1-homework2.py：编写函数判断素数(数字)，返回 True/False。在主程序中输入数字并调用函数判断

挑战作业： 3. 新建 2-1-homework3.py：实现一个带函数封装的"石头剪刀布"游戏：

- 获取用户选择() → 返回"石头"/"剪刀"/"布"
- 电脑随机出拳() → 返回电脑选择
- 判断胜负(玩家, 电脑) → 返回胜负结果
- 主程序循环对战，直到玩家输入"退出"

洛谷专项训练： https://www.luogu.com.cn/problem/P5735
https://www.luogu.com.cn/problem/P5737
https://www.luogu.com.cn/problem/P5739
https://www.luogu.com.cn/problem/P1304

本小节小结

通过 2.1 节的学习，你掌握了**函数**这一神器：

- **定义函数**：`def 函数名(参数):`
- **调用函数**：`函数名(参数)`
- **返回值**：`return` 把结果送回来
- **作用域**：局部变量和全局变量的区别

记住：**函数是程序员的乐高积木**，用得好能让代码从"一摊烂泥"变成"精美建筑"。现在你可以把重复代码打包，让程序变得简洁、优雅、易于维护！

2.2 函数进阶

2.2.1 *匿名函数 lambda

有时候我们需要一个**简单函数**，但懒得写 def 和起名字。就像点外卖时选择"匿名用户"评价一样，**lambda** 就是函数的"匿名模式"。

基本语法：

```
# 格式：lambda 参数：表达式
lambda x: x * 2  # 这是一个函数对象，功能：输入 x，返回 x*2

# 但这样写没法用，要赋值给变量
翻倍 = lambda x: x * 2
print(翻倍(5))  # 输出：10
```

lambda limitations：

只能写一个表达式，不能写语句（如 if、for、赋值）

自动返回表达式的结果，不需要 return

适合简单逻辑，复杂逻辑还是要用 def

经典场景：排序 key 函数

```
students = [
   {"name": "小明", "score": 85},
   {"name": "小红", "score": 92},
   {"name": "小刚", "score": 78}
]

# 按分数排序，lambda 告诉 sorted 用每个元素的['score']作为排序依据
students.sort(key=lambda s: s["score"], reverse=True)
print(students)  # 小红、小明、小刚
```

生活比喻：

- **def 函数**：正式员工，有工牌、有全名，能干复杂工作
- **lambda**：临时工，没名字（匿名），只干简单的活儿

2.2.2 *高阶函数：函数当参数用

在 Python 里，函数是**一等公民**，可以像变量一样传来传去。**高阶函数**就是接收函数作为参数的函数。

map()：批量加工

```
# 把列表每个元素都变平方
numbers = [1, 2, 3, 4, 5]
平方列表 = map(lambda x: x**2, numbers)  # map 返回迭代器
print(list(平方列表))  # 输出: [1, 4, 9, 16, 25]
```

filter()：批量筛选

```
# 筛选偶数
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
偶数 = filter(lambda x: x % 2 == 0, numbers)  # 保留使函数返回 True 的元素
print(list(偶数))  # 输出: [2, 4, 6, 8]
```

sorted()：自定义排序

```
# 按字符串长度排序
words = ["apple", "hi", "watermelon", "cat"]
长单词在前 = sorted(words, key=len, reverse=True)
print(长单词在前)  # 输出: ['watermelon', 'apple', 'cat', 'hi']
```

现代趋势：列表推导式更香 虽然高阶函数很酷，但 Python 程序员更喜欢列表推导式，因为可读性更强：

```
# map 写法
map(lambda x: x*2, numbers)

# 列表推导式写法（推荐）
[x*2 for x in numbers]

# filter 写法
filter(lambda x: x>3, numbers)

# 列表推导式带条件（推荐）
[x for x in numbers if x>3]
```

2.2.3 递归函数：自己调用自己

递归就像俄罗斯套娃，或者查词典时用词解释词。函数内部调用自身，直到满足**终止条件**。

经典例子：阶乘

```
def 阶乘(n):
    """n! = n * (n-1) * ... * 1"""
    # 终止条件：1 的阶乘是 1
    if n == 1:
        return 1
    # 递归调用：n! = n * (n-1)!
    return n * 阶乘(n - 1)

print(阶乘(5))  # 5 * 4 * 3 * 2 * 1 = 120
```

执行过程拆解（n=3 时）：

```
阶乘(3)
→ 3 * 阶乘(2)
→ 3 * (2 * 阶乘(1))
→ 3 * (2 * 1)  # 遇到终止条件
→ 3 * 2
→ 6
```

递归三要素（必背）：

1. **终止条件**：什么时候停止（`if n==1`）
2. **递归公式**：大问题如何变小（`n! = n * (n-1)!`）
3. **向终止条件靠拢**：每次调用都更接近终止条件（`n-1`）

生活比喻：

- **查字典**：查"递归"→见"递归"→见"递归"...（死循环，没终止条件）
- **数鸡群**：想知道有多少只鸡→数一只，剩下的让朋友数→朋友也数一只，再让朋友的朋友数...→最后一只不用再数

慎用递归！ Python 默认递归深度限制 1000 层，太深会**栈溢出**。能用循环尽量用循环：

```
# 递归（简洁但危险）
def 斐波那契(n):
```

```
    if n <= 2:
        return 1
    return 斐波那契(n-1) + 斐波那契(n-2)

# 循环（安全且高效）
def 斐波那契循环(n):
    a, b = 1, 1
    for _ in range(n-1):
        a, b = b, a+b
    return a
```

2.2.4 函数也是对象

在 Python 里，**函数名就是变量**，指向函数对象本身。

```
def 你好():
    print("你好世界")

print(你好)  # 输出: <function 你好 at 0x...>
print(type(你好))  # 输出: <class 'function'>

# 把函数赋值给别的变量
问候函数 = 你好
问候函数()  # 输出: 你好世界（效果完全一样）

# 函数存进列表
函数列表 = [lambda x: x+1, lambda x: x*2, lambda x: x**2]
for func in 函数列表:
    print(func(5))  # 分别输出: 6, 10, 25

# 函数当返回值
def 选择运算(操作符):
    if 操作符 == "+":
        return lambda x, y: x + y
    elif 操作符 == "*":
```

```
        return lambda x, y: x * y

加法函数 = 选择运算("+")
print(加法函数(3, 4))  # 输出: 7
```

比喻： 函数就像机器，函数名是机器的遥控器。你可以把遥控器复制一份给别人，也可以把机器放进仓库（列表）里按需取用。

2.2.5 闭包：函数记住外部状态

闭包是函数进阶的里程碑概念：一个函数内部定义另一个函数，内部函数**记住**了外部函数的变量。

```
def 创建计数器(初始值=0):
    count = 初始值  # 外部函数的局部变量

    def 计数器():
        nonlocal count  # 声明要修改外部变量
        count += 1
        return count

    return 计数器

# 创建两个独立的计数器
计数器A = 创建计数器(10)
计数器B = 创建计数器(100)

print(计数器A())  # 输出: 11
print(计数器A())  # 输出: 12
print(计数器B())  # 输出: 101（互不影响）
```

闭包三要素：

1. **嵌套函数**：内部函数计数器()
2. **外部变量**：内部函数使用了 count
3. **返回内部函数**：return 计数器

生活比喻：

- **旅行青蛙**：每次出门（调用）都带着上一次旅行的记忆（状态）

•**银行账户**：每个账户（闭包）记住自己的余额（状态），互不干扰

2.2.6 *装饰器初探（选学）

装饰器是**高阶函数+闭包**的集大成者，用于在不修改原函数代码的情况下，给函数增加功能。

```
def 计时器(原函数):
    """装饰器：记录函数执行时间"""
    import time

    def 包装函数(*args, **kwargs):  # *args 接收所有位置参数
        开始 = time.time()
        结果 = 原函数(*args, **kwargs)
        结束 = time.time()
        print(f"{原函数.__name__}执行了{结束-开始:.4f}秒")
        return 结果

    return 包装函数

# 使用@语法糖应用装饰器
@计时器
def 慢函数():
    """模拟耗时操作"""
    sum(i for i in range(1000000))

慢函数()  # 自动打印执行时间
```

执行逻辑：

1. @计时器 等价于 慢函数 = 计时器(慢函数)
2. 调用慢函数()时，实际执行的是包装函数()
3. 包装函数里调用了真正的原函数，并添加了计时功能

```
内置装饰器：@property 把方法变成属性调用：
class 圆:
    def __init__(self, 半径):
        self.半径 = 半径
```

```
    @property
    def 面积(self):
        return 3.14159 * self.半径 ** 2

c = 圆(5)
print(c.面积)  # 调用方法像访问属性一样，不用加括号()
```

注意： 装饰器是进阶内容，初学者先理解概念，不必立即掌握。

2.2.7 函数进阶综合实践

案例：简易计算器（带历史记录）

```
# 2_2_7_calculator.py
历史记录 = []  # 全局变量

def 记录(表达式, 结果):
    """闭包式历史记录"""
    def 保存():
        历史记录.append(f"{表达式} = {结果}")
        print("已记录")
    return 保存

def 计算器():
    """主计算器函数"""
    print("=== 简易计算器 ===")
    print("支持: + - * / , 输入'退出'结束")

    while True:
        输入 = input(">>> ")
        if 输入 == "退出":
            break

        try:
            # 危险! eval 可执行任意代码，仅教学使用
```

```
            结果 = eval(输入)
            print(f"结果: {结果}")

            # 使用闭包记录
            保存函数 = 记录(输入, 结果)
            保存函数()
        except:
            print("表达式错误! ")

    print("\n=== 计算历史 ===")
    for i, 记录项 in enumerate(历史记录, 1):
        print(f"{i}. {记录项}")

计算器()
```

运行效果：

```
=== 简易计算器 ===
支持: + - * / , 输入'退出'结束
>>> 3 + 5
结果: 8
已记录
>>> 10 * 2
结果: 20
已记录
>>> 退出

=== 计算历史 ===
1. 3 + 5 = 8
2. 10 * 2 = 20
```

课后作业

基础作业：

1. 新建 2-2-homework1.py：用 lambda 和 sorted，将学生列表按成绩降序排序

```
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78)]
# 输出: [("Bob", 92), ("Alice", 85), ("Charlie", 78)]
```

2. 新建 2-2-homework2.py：编写递归函数计算斐波那契数列第 n 项，用循环版本对比效率

挑战作业： 3. 新建 2-2-homework3.py：实现缓存装饰器，让函数记住之前计算过的结果（**记忆化**）

```
@缓存
def 斐波那契(n):
    if n <= 2:
        return 1
    return 斐波那契(n-1) + 斐波那契(n-2)

# 第一次计算斐波那契(35)慢，第二次瞬间返回
```

洛谷专项训练： https://www.luogu.com.cn/problem/P5739

本小节小结

通过 2.2 节的学习，你迈入了函数的高级殿堂：

- **lambda**：简洁的匿名函数
- **高阶函数**：函数当参数用（map/filter/sorted）
- **递归**：函数自己调用自己（注意终止条件）
- **函数对象**：函数名就是变量
- **闭包**：函数记住外部状态
- **装饰器**：给函数加功能（高阶+闭包的组合拳）

核心心法：

1. **简洁优先**：lambda 和推导式让代码优雅，但别为了炫技牺牲可读性
2. **递归慎用**：能用循环就用循环，除非问题本身适合递归（如树遍历）
3. **闭包是宝藏**：理解闭包，才能理解装饰器和回调函数
4. **装饰器是框架基石**：Django/Flask 大量用装饰器，先理解思想

现在你可以写出更抽象、更复用、更"Pythonic"的代码了！下一节将学习**模块**，把函数组织成更大的武器库。

感谢您看到这里，这本书在这里就结束了，下面将会分成两条线：

1：大厂工程代码线

学习更加健壮和利于维护的代码，复杂度不高，但是需要长期维护

2：信息竞赛冲击线

学习更高级更专业的代码，凹运行时长和最优算法，甚至超越人工智能。