

福州大学超算团队

AI 方向

学习手册

I. python 入门

September 06, 2024

Contents

I. Python 简介	5
I.1. 开发环境配置	5
I.1.1. python 安装	5
I.1.2. 环境变量	5
I.1.3. 编辑器	5
I.2. 编写你的第一个程序	6
II. Python 基础语法	7
II.1. 数据类型、变量与基本运算	7
II.1.1. Python 数据类型	7
II.1.1.1. 整数	7
II.1.1.2. 浮点数	7
II.1.1.3. 字符串	8
II.1.1.4. 布尔值	9
II.1.1.5. 空值	10
II.1.2. 变量	10
II.1.3. 基本运算及其优先级	12
II.2. 分支	14
II.3. 循环	16
II.4. 过程抽象	18
II.4.1. 函数	18
II.5. 数据抽象	21
II.5.1. 类与对象	21
II.5.1.1. 使用类	21
II.5.1.2. 创建类	21
III. 数据结构与标准库	24
III.1. 数据结构与 ADT	24
III.2. List	24
III.3. Tuple	27
III.4. Dict	28
III.5. Set	30
III.6. Algorithms	32
III.6.1. 排序	32
III.6.2. 查找最大值和最小值	33
III.6.3. 求和与平均值	34
III.6.4. 过滤和映射	34
III.6.5. 集合操作	35
III.6.6. 字符串操作	35
III.7. 模块	36
III.7.1. math	37
III.7.2. time	37

III.7.3. datetime	38
III.7.4. random	38
III.7.5. os	38
IV. 编程范式入门	40
IV.1. 面向过程编程	40
IV.2. 面向对象编程	40
IV.2.1. 面向对象三大特性	43
IV.2.1.1. 封装 Encapsulation	43
IV.2.1.2. 继承 Inheritance	45
IV.2.1.3. 多态 Polymorphism	46
IV.3. 函数式编程	48
IV.4. 元编程	52
IV.4.1. 反射	53
IV.4.2. 泛型	53
V. 工具链	56
V.1. 调试器	56
V.1.1. pdb 常用命令	56
V.2. 包管理	57
V.2.1. pip 简单操作	58
V.2.2. 更换库源	58
V.3. 虚拟环境管理	58
V.3.1. venv	59
V.3.2. conda	59
V.3.2.1. conda 常用操作	60

前言

编写这本学习手册的目的在于，让没什么编程经验的小白也能比较好的入门。我们尝试用比较容易理解的方式向大家介绍编程中的一些重要概念和思想，这些都是从我们自身的学习过程中得到的经验和思考，当然也会因为水平有限，有些地方写的并不“通俗”，如果你有任何建议，请向我们反馈。

在手册中，你会看到下面三种模块，与正文密切相关，请不要忽略

任务

该模块中内容是一些需要大家完成的任务，用于巩固所学

注意

该模块的内容是一些注意事项，如果你觉得看不懂正文或者遇到解决不了的问题，可以参考这部分内容

建议

该模块的内容是一些建议，包括思考的方式，工具的推荐，都是我们宝贵的经验

本文的一些连接是CSDN的网站教程，并不说明CSDN是最佳的选择，但考虑到刚入门可能大多数的教程都来源于CSDN，同时考虑网络的原因，我们给出了一些这样的教程连接，如果有可能，请使用更优质的网站

下面开始学习吧

I. PYTHON 简介

I.1. 开发环境配置

I.1.1. PYTHON 安装

到[官方网站](#)下载根据你的系统选择对应版本的 Python 解释器(Windows,Linux,Macos)，推荐 python 版本为 3.9 以上。具体安装过程可以参考[Python 安装教程](#)

注意事项

- » 在项目中，绝大多数的路径都是英文字符的，常常会因为中文路径而产生一些未知的问题，因此建议之后安装的软件统一使用全英文的路径。
- » 若使用的系统是 Windows，建议下载路径不要选择 C 盘(若只有 C 盘或者 C 盘空间足够大可以忽略)。原因在于 C 盘是默认的系统盘，里面存储的大部分文件是系统所需，若 C 盘空间不足，一定程度上会影响体验。

在 Python 安装完成后，在你所选择的安装目录下，就会有 `python.exe` 文件，点击即可运行 python 执行器，然后通过键盘输入 `1+1`，按下回车之后，可以看到对应的计算结果为 `2`。上面的例子就是一个非常简单的程序了，但是这样的方式是极其不便的，只能完成一些简单的程序。后面我们会介绍通过在代码编辑器中编写程序，以文件的形式运行程序的方法。

I.1.2. 环境变量

什么是环境变量？请先在 Python 的安装目录下打开终端

建议

- » 当你不知道怎么做的时候，就要去网上搜寻相关的资料了，我们并不总会告诉你该如何做，必要的时候，要自行去搜索相关的资料完成相应的操作，通过这样的方式来不断提高解决问题的能力。
- » 搜索引擎的选择：Edge 浏览器自带的 Bing, Chrome 的 google(需科学上网)
- » 在提问之前，记住原则：1. STFW、2. RTFSC、3. RTFM (Search the friendly web, Read the friendly source code, read the friendly manual)

在当前目录打开终端后，你会发现终端同样也是一个交互式界面，通过键盘输入 `./python.exe`，同样可以运行 python 执行器。上面的操作中，`./` 代表的是在当前目录，而 `./python.exe` 则代表在当前目录下执行 `python.exe` 这个可执行文件。

如果我们希望在任何目录下，通过键入 `python.exe` 即可执行 python 解释器呢？即不通过指定执行器所在的目录的方式就可以运行，这个时候就需要我们去配置环境变量了。环境变量的作用之一就是：告诉操作系统你的可执行文件的路径。具体配置 Python 环境变量的方法参考[Win11 中 Python 环境变量配置教程](#)

I.1.3. 编辑器

推荐使用 VS Code 作为代码编辑器，到[VS Code 官网](#)下载安装包。Vs code 是一个强大的编辑器，可定制化性较高，插件系统非常完备，更多的玩法，自行探索。具体安装可以参考[VS code 安装教程](#)

I.2. 编写你的第一个程序

在 Visual Studio Code 中，项目的单位是文件夹，即工作区。在工作区中有多个文件，通过点击文件可以在编辑窗口中编辑文件。

新建项目

- » 创建新文件夹
- » 使用 VSCode 打开该文件夹
- » 在该文件夹下创建 main.py 文件
- » 在你的 main.py 文件中，输入 `print("hello world!")`，然后保存

在 VSCode 中可以用使用快捷键 `ctrl + ~` 唤出终端，在终端键入 `python main.py`，回车之后，你的第一个程序就运行起来了！那么你能理解这段代码的含义吗？`print` 就是一个在终端打印输出的函数，至于什么是函数，会慢慢讲到的。

到这里，开发环境已经准备好了。接下来我们来介绍 Python 的基础语法和代码结构

II. PYTHON 基础语法

II.1. 数据类型、变量与基本运算

II.1.1. PYTHON 数据类型

程序离不开数据。把数字、字母和文字输入到计算机，就是希望它利用这些数据完成某些任务。接下来我们就来介绍在 Python 中常用的数据类型。

II.1.1.1. 整数

整数，就是没有小数的数据类型，这和数学上的整数概念是一致的。在 Python 中，整数的范围理论上是无限的，我们称之为高精度。Python 3 通过其内置的 `int` 类型，支持任意精度的整数，这意味着你可以表示和处理非常大的整数，唯一的限制是可用的内存和计算资源。

```
a = 1234567890123456789012345678901234567890  
b = 9876543210987654321098765432109876543210  
result = a + b  
print(result)
```

试着运行一下看看

将上面的代码复制下来运行一下

从结果上来看，无论数字多大，Python 都能够正确处理。这与 C 语言不同，后者通常有固定的整数范围，例如 `int` 或 `long` 类型。对于很大的数，例如 `10000000000`，很难清楚 0 的个数。Python 允许我们使用 `_` 对整型进行分隔。写成 `10_000_000_000` 和 `10000000000` 是完全一样的。

II.1.1.2. 浮点数

浮点数也就是小数，之所以称为浮点数，是因为按照科学记数法表示时，一个浮点数的小数点位置是可变的。比如， 1.20×10^9 和 12.0×10^8 是完全相等的。以下是关于 Python 浮点数的重要特点。

1. 浮点数表示

- » 对于很大或者很小的浮点数，必须使用科学计数法表示，把 10 用 e 替代，比如 1.2×10^9 和 `1.2e9` 等价。
- » 浮点数能表示的范围是从 `1.7e-308` 到 `1.7e+308`，但精度有限，一般为 15-17 位有效数字。

```
large_num = 1.23e5 # 1.23 * 10^5, 输出 123000.0  
small_num = 1.23e-5 # 1.23 * 10^-5, 输出 0.0000123
```

2. 浮点数精度

浮点数的精度有限，因此在处理非常大或非常小的数时，可能会出现舍入误差。例如：

```
x = 0.1 + 0.2  
y = 1.7e+308  
print(x)  
print(2*y)  
# 输出 0.30000000000000004  
# 输出 inf
```

试着运行一下看看

尝试运行上诉代码，观察输出的结果是否与你预想中的不太一样

由于浮点数的二进制表示限制，无法精确表示某些小数，这种误差通常在科学计算中很常见。可能你不太理解这方面的内容，目前我们不需要过分深入，在未来学习 C 语言内存管理时，会深入讨论关于二进制、内存管理方面的知识。

int 和 float

你可能会疑惑，既然有了 float，为什么需要 int 类型呢？在运算上完全可以用 float 替代 int，事实上 float 和 int 在计算机中的存储的方式并不一样（虽然都是二进制），有些功能必须由 int 来完成

II.1.1.3. 字符串

Python 的字符串是用于表示文本的数据类型。它是不可变的序列类型，这意味着一旦创建，字符串的值就不能修改。字符串可以用单引号 '、双引号 "、三引号 """ 或 """ 来定义：

1. 字符串的定义

```
# 单引号
str1 = 'Hello'

# 双引号
str2 = "World"

# 三引号 (可用于定义多行字符串)
str3 = """This is
a multiline
string""
```

注意，' 和 " 只是一种表达方式，并不作为字符串的一部分。所以 "Hello" 只有 H, e, l, l, o 这 5 个字符。如果 ' 本身也是一个字符的话，那就可以使用 ""，比如 "I'm Superman"。

2. 转义字符

如果字符串内部既包含'又包含"怎么办？可以用转义字符 \ 来标识，比如：

```
x = 'I\m \"Superman\"!'
print(x)
# 输出： I'm "Superman!"
```

动手试试

转义字符 \ 可以转义很多字符，比如 \n 表示换行，\t 表示制表符，字符 \ 本身也要转义，所以 \\ 表示的字符就是 \。

大家请自己尝试创建几个字符串去 print() 看看结果。

如果字符串里面有很多字符都需要转义，就需要加很多 \，为了简化，Python 还允许用 r" 表示 " 内部的字符串默认不转义。比较一下下列的输出：

```
print("\\\\t\\\\")  
print(r'\\\\t\\\\')
```

关于字符串

字符串的操作贯穿了 Python 的学习，在此只是简单的介绍字符串类型，在后续的章节会详细介绍字符串的相关操作。

II.1.1.4. 布尔值

在 Python 中，布尔(Boolean)是一种表示逻辑值的数据类型，主要用于控制程序的流程和判断逻辑。布尔类型只有两个值：`True` 和 `False`，分别代表逻辑上的“真”和“假”。

1. 布尔类型的表示

布尔值在 Python 中由关键字 `True` 和 `False` 表示（注意大小写）。布尔值是 `bool` 类型的实例。

```
a = True  
b = False
```

2. 布尔运算

» 与运算（`and`）：只有当两个操作数都为 `True` 时，结果才为 `True`。

```
True and True # 结果是 True  
True and False # 结果是 False
```

» 或运算（`or`）：只要有一个操作数为 `True`，结果就是 `True`。

```
True or False # 结果是 True  
False or False # 结果是 False
```

» 非运算（`not`）：取反运算，将 `True` 变为 `False`，将 `False` 变为 `True`。

```
not True # 结果是 False  
not False # 结果是 True
```

3. 比较运算符

比较运算符用于比较两个值的大小或相等性，结果为布尔类型。

» 等于（`==`）：判断两个值是否相等。

» 不等于（`!=`）：判断两个值是否不相等。

» 大于（`>`）、小于（`<`）、大于等于（`>=`）、小于等于（`<=`）：

```
5 == 5 # 结果是 True  
5 == 6 # 结果是 False
```

```
5 != 6 # 结果是 True
```

```
5 != 5 # 结果是 False
```

```
5 > 3 # 结果是 True  
3 < 2 # 结果是 False  
4 >= 4 # 结果是 True  
3 <= 1 # 结果是 False
```

II.1.1.5. 空值

空值是 Python 里一个特殊的值，用 `None` 表示。`None` 不能理解为 0，因为 0 是有意义的，而 `None` 是一个特殊的空值。此外，Python 还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

II.1.2. 变量

在 Python 中，变量是用于存储数据的容器。也就是装数据的瓶子，变量在 Python 中不需要声明其类型，Python 是一种动态类型语言，因此可以自动推断变量的类型(整数、浮点数、字符串等等)。

1. 变量的命名

- » 变量名必须以字母 (a-z 或 A-Z) 或下划线 `_` 开头，后面可以跟字母、数字或下划线。
 - » 变量名是区分大小写的 (`myVar` 和 `myvar` 是两个不同的变量)。
 - » 变量名不能是 Python 的保留关键字 (如 `if`, `else`, `for` 等)。
-

```
# 正确的命名  
my_variable = 10  
_myVar = "Hello"  
  
# 错误的命名会抛出 SyntaxError 错误  
3myvar = 15
```

变量的正确命名是一件大事

正确且恰当的变量命名可以让工程结构更加清晰化，为编程便利提供帮助。当变量增多时，多使用下划线 `_` 为变量完整命名。变量的命名规范可参考 [变量命名规范](#)

2. 变量的赋值

在 Python 中，变量通过赋值语句 (`=`) 将值赋给变量，并且允许在一行代码中同时为多个变量赋值。

```
x = 5  
y = "Hello, World!"  
x, y, z = 1, 2, "Python" # 多变量赋值
```

尝试一下

你可以自己动手测试一下，`x`、`y`、`z` 内储存的内容。

可以创建变量当然也意味着可以删除。使用 `del` 关键字可以删除变量。

```
x = 10  
del x # 变量 x 被删除
```

3. 变量的类型

Python 中每个变量都有一个类型，你可以使用 `type()` 函数来查看变量的类型。

```
x = 10      # int  
y = 3.14     # float  
z = "Python" # str  
is_true = True # bool  
print(type(x)) # <class 'int'>
```

由于 Python 是动态类型语言，变量的类型可以在运行时更改。例如：

```
x = 10      # x 是整数类型  
x = "Hello" # x 现在变成字符串类型
```

4. 全局变量和局部变量

在函数中定义的变量是局部变量，只在函数内部可见。如果需要在函数外部访问变量，可以使用 `global` 关键字定义全局变量。这方面内容不需要现在了解，在后续函数的章节会着重介绍。

5. 常量

虽然 Python 没有原生的常量（值不会改变的变量），但是通常使用全大写字母命名变量来表示它是常量，比如：

```
PI = 3.14159
```

小测试

理解变量在计算机内存中的表示也非常重要。当我们写

```
a = 'QWE'
```

Python 的解释器做了两件事：

1. 在内存中创建了一个 'ABC' 的字符串；
2. 在内存中创建了一个名为 a 的变量，并把它指向 'ABC'。

也可以把一个变量 a 赋值给另一个变量 b，这个操作实际上是把变量 b 指向变量 a 所指向的数据。

```
a = 'QWE'  
b = a  
a = 'ABC'  
print(b)
```

最后一行打印出变量 b 的内容到底是 'ABC' 呢还是 'XYZ'？ Try it!!!!!!

II.1.3. 基本运算及其优先级

Python 提供了多种基本运算，包括算术运算、比较运算、逻辑运算等。了解这些运算符及其优先级对编写正确的程序非常重要。以下是 Python 中的基本运算符及其优先级规则。

1. 算术运算符

Python 支持常见的算术运算：

运算符	描述	示例
+	加法	<code>3 + 2 =5</code>
-	减法	<code>3 - 2 =1</code>
*	乘法	<code>3 * 2 =6</code>
/	除法	<code>3 / 2 =1.5</code>
//	整除	<code>3 // 2 =1</code>
%	取余	<code>3 % 2 =1</code>
**	幂运算	<code>3 ** 2 =9</code>

```
a = 10
b = 3
print(a + b) # 输出 13
print(a - b) # 输出 7
print(a * b) # 输出 30
print(a / b) # 输出 3.3333...
print(a // b) # 输出 3
print(a % b) # 输出 1
print(a ** b) # 输出 1000
```

动手尝试

动手尝试一下 Python 的算术运算

2. 比较运算符

比较运算符用于比较两个值，并返回布尔值（True 或 False）：

运算符	描述	示例
==	等于	<code>3 == 3 =True</code>
!=	不等于	<code>3 != 2 =True</code>
>	大于	<code>3 > 2 =True</code>
<	小于	<code>2 < 3 =True</code>
>=	大于等于	<code>3 >= 2 =True</code>
<=	小于等于	<code>2 <= 3 =True</code>

```
a = 10
b = 5
print(a > b) # 输出 True
print(a == b) # 输出 False
```

3. 逻辑运算符

运算符	描述	示例
and	与	True and False = False
or	或	True and False = True
not	非	not False = True

```
a = True
b = False
print(a and b) # 输出 False
print(a or b) # 输出 True
print(not a) # 输出 False
```

4. 赋值运算符

Python 的赋值运算符用于将值赋给变量，可以与算术运算符结合使用。

运算符	描述	示例
=	赋值	x = 5
+=	加法赋值	x += 5 等同于 x = x + 5
-=	减法赋值	x -= 5 等同于 x = x - 5
*=	乘法赋值	x *= 5 等同于 x = x * 5
/=	除法赋值	x /= 5 等同于 x = x / 5
//=	整除赋值	x //= 5 等同于 x = x // 5
%=	取余赋值	x %= 5 等同于 x = x % 5
**=	幂赋值	x **= 5 等同于 x = x ** 5

```
x = 10
x += 5 # x 现在是 15
x *= 2 # x 现在是 30
```

5. 位运算符

位运算符用于对整数进行二进制操作。如果你现在还不是很了解什么是二进制，可以先去理解一下二进制。

运算符	描述	示例

&	按位与	$3 \& 2 = 2$
	按位或	$3 2 = 3$
^	按位异或	$3 ^ 2 = 1$
~	按位取反	$\sim 3 = -4$
<<	按位左移	$3 << 1 = 6$
>>	按位右移	$3 >> 1 = 1$

```
a = 3 # 011
b = 2 # 010
print(a & b) # 输出 2, 二进制为 010
```

6. 运算优先级

在数学当中，我们都知道乘法和除法要先算，后算加减，在编程语言中，同样存在这样的运算规则，这样子先算某一种运算符再算下一种运算符暗喻着不同种运算符存在一种优先顺序，我们称之为优先级。Python 运算符的优先级决定了运算的顺序。优先级从高到低如下：

- » 指数运算符： **
- » 按位取反： ~
- » 乘法、除法、整除、取余： *， /， //， %
- » 加法、减法： +， -
- » 位移运算： <<， >>
- » 按位与： &
- » 按位异或： ^
- » 按位或： |
- » 比较运算： <， <=， >， >=， !=， ==
- » 逻辑非： not
- » 逻辑与： and
- » 逻辑或： or
- » 赋值运算： =， +=， -=， *=， /=， %=， **=， //=， &=， |=， ^=， >>=， <<=

如果想改变默认的运算顺序，可以使用括号 () 提高优先级：

```
result = (2 + 3) * 4 # 结果是 20，因为括号内的加法先计算
```

II.2. 分支

计算机在面对不同场景的时候一样也有自己的判断，这也是为什么它能完成许多自动化任务的原因。这章节我们就来了解一下，如何为计算机植入能做出判断的“大脑”。

1. if 语句

比如，我们要判断一个数字是否大于 5，我们就可以使用 if 进行判断。

```
x = int(input("请输入你要验证的数字"))
if x > 5:
    print("x is greater than 5")
```

解释

if 相关的判断语句常与布尔类型的值挂钩，当 `x > 5` 的条件为真时，才执行 `print("x is greater than 5")`。

关于 I/O

什么是 I/O？简单来说就是计算机的输入与输出。在 Python 中分别对应 `input()` 和 `print()`，`input()` 用来接受来自你的输入，而 `print()` 则是将你指定的内容打印到屏幕上。

关于缩进

我们观察到在使用 if 条件后需要 :，并且在 `print()` 前需要缩进（通常使用四个空格）。与许多其他编程语言不同，Python 不使用花括号 {} 来定义代码块，而是依赖缩进来表示代码的层次结构。Python 非常依赖缩进来确定代码的逻辑结构，错误的缩进会导致 `IndentationError` 错误。

2. else 语句

在上面的例子中，我们处理 `x > 5` 的情况。现在，我想对剩下的情况也进行一个输出，我们该怎么做呢？这就需要用到 `else` 语句了。

```
x = int(input("请输入你要验证的数字"))
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

3. elif

`elif` 是 "else if" 的缩写。在学习数学区间的时候，我们并不是只把一个进行简单的一刀切划分成两个区间。面对多个区间的分支，我们就需要使用到 `elif` 语句。就用我们用年龄划分为例：

```
age = 15
if age >= 18:
    print("你已经成年了")
elif age >= 12:
    print("你是青少年")
else:
    print("你是小孩子")
```

当 `age` 是 15 时，第一个 `if` 条件为假，但 `elif` 条件为真，因此执行“你是青少年”的打印语句。

if 语句执行的特点

`if` 语句执行有个特点，它是从上往下判断，如果在某个判断上是 `True`，把该判断对应的语句执行后，就忽略掉剩下的 `elif` 和 `else`。

所以你可以解释一下为什么下面程序打印的是“你是青少年”吗？

```
age = 20
if age >= 6:
    print('你是青少年')
elif age >= 18:
    print('你是成年人')
else:
    print('你还是个孩子')
```

3. 单行条件表达式(三目运算符)

这是什么东西？简单来说，我们只需要对区间进行一刀切分成两个区间时，再去用代码块的形式会显得代码很冗余且不够简洁。Python 允许使用单行的条件表达式（也叫三元表达式）来简化代码：

```
age = 20
message = "成年人" if age >= 18 else "未成年人"
print(message)
```

任务

» 使用三元表达式实现：两个数取较大数

II.3. 循环

循环，意思就是不断的重复做一件事。现在让我们来计算一道数学题 $1+2+3+\dots+10000$ 把它交给计算机，这很简单。那如果是 $1+2+3+\dots+10000$ 呢？直接把它交给计算机怕是不可能了。为了能让计算机得到这么庞大的表达式，我们就需要使用循环。

1. for 循环

`for` 循环通常用于遍历序列（如列表、元组、字符串、字典或集合）。它会在序列的每个元素上循环一次。

```
for i in range(5):
    print(i)
# 输出将是从 0 到 4 的数字。
```

`range()` 函数是 Python 中常用来生成数值序列的函数，常与 `for` 循环一起使用。

动手尝试一下

你可以在自己的计算机上体验一下 `range()` 函数的魅力，并完成计算 1-100 整数之和。

```
sum = 0
for x in [...]:
    sum += x
print(sum)
```

请在 [...] 处填入相关代码。

关于序列

序列的相关知识点已经出现在第三章数据结构中，在这里先感受一下循环的魅力

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

`for` 循环可以遍历序列内的内容并对他们进行操作

2. while 循环

`while` 循环，作为循环的一种，但是不如 `for` 来的灵活且常用。

```
count = 0
while count < 5:
    print(count)
    count += 1
```

`count += 1` 作为控制循环结束的条件，如果没有这段代码，`while` 循环将不会停止。

3. 控制循环的语句

`continue`：试想一下，当我们对一组数据进行打印的时候出现了一个我们不想要打印的数字，我们该如何避免它呢？那我们肯定想要跳过它！

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

`continue` 为我们提供了这个需求，它可以帮我们跳过当前的迭代，继续下一次的迭代。也就是说当我们遇到 3 的时候，我们不会去执行 `print()`。输出为 0, 1, 2, 4。

`break`：同样的，当一组数据打印到某个地方，我们就想要它停止打印。

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

`break` 为我们提供了打断的需求。输出为 `0, 1, 2`，因为在 `i` 等于 `3` 时，`break` 终止了循环。

`else` 子句：`for` 和 `while` 循环可以带有 `else` 子句。当循环正常结束时，`else` 子句的代码会被执行；如果循环被 `break` 终止，`else` 代码块则不会被执行。

```
for i in range(5):
    print(i)
else:
    print("循环正常结束")
```

输出为 `0, 1, 2, 3, 4` 以及 “`循环正常结束`”。

再议 `input()`！

分支和循环的知识点结束，皆大欢喜。但是在这里我还是想再说一说 `input()` 这个函数。在前面我们使用到 `input()` 的例子中，我们还使用了 `int()`。这是为什么呢？大家可以试试把 `int()` 删除，看看会出现什么错误！

在这里我也不给大家卖什么关子，`input()` 返回的数据类型是 `str`，也就是说无论你输入的是什么数字还是字母，计算机都会自动把它认为是字符串类型。所以我们要使用 `int()` 这个内置的函数，将 `str` 类型转化为 `int`。这样整数才能和整数进行比较对吧！

II.4. 过程抽象

过程抽象（Procedural Abstraction）是编程中的一种设计思想，它指的是将一系列具体的操作步骤或实现细节隐藏在一个过程、函数或方法的背后。简单来说，我们只需要了解这个过程的输入和输出，而不需要关心其具体实现。

II.4.1. 函数

现在我们来定义一个函数 $f(x) = x + 1$ ，意思是输入一个数，得到的输出比输入大 1。我们可以多次使用它来得到输出，并且在条件允许的情况下对它进行修改。所以学习函数的重点在于提高代码的复用性和可维护性。

1. 调用函数

Python 内置了很多有用的函数，我们可以直接调用。比如 `print()` 和 `input()` 也是属于函数的类型，我们可以多次调用它们。

Python 文档

要调用一个函数，需要知道函数的名称和参数。如 `print()` 函数，`print` 是函数名，`()` 内输入的内容是函数的参数。一般的函数都不仅仅只有一个参数，具体可以通过 Python 官方文档查阅 <https://docs.python.org/3/library/functions.html>

2. 定义函数

在 Python 中，定义一个函数要使用 `def` 语句，依次写出 函数名、括号、括号中的参数 和冒号`:`，然后，在缩进块中编写函数体，函数的返回值用 `return` 语句返回。

```
def greet(name):
    print(f"Hello, {name}!")
    print("How are you?")
greet("Alice")
# 输出:
# Hello, Alice!
# How are you?
```

正如你所看到的那样，`greet` 函数封装了打印问候信息的过程。使用者只需调用 `greet`，而不必关心 `print` 的具体实现。

如果你只是想要先定义一个函数，但不具体实现它的功能。可以定义一个空函数：

```
def nop():
    pass
```

缺少了 `pass`，代码运行就会有语法错误。

2. 函数参数与返回值

- » 函数的参数允许我们根据不同的输入，灵活执行相同的操作，从而避免重复代码。参数在函数定义时指定，调用时传入实际值。
- » 函数可以返回结果，也可以不返回值。返回值通过 `return` 语句实现，调用者可以获取和使用这些返回的结果。

```
def add_and_sub(a, b):
    return a + b, a - b

a, b = add(3, 5)
print(a)
print(b)
# 输出: 8
#     -2
```

在这个例子中，`add_and_sub` 函数接收两个参数 `a` 和 `b`，并返回它们的和与差。调用者可以获得这个返回值并进行进一步处理。

关于两个位置的 `a,b`

在函数内的 `a`，`b` 为函数的形式参数，函数外的 `a`，`b` 为程序的变量。两者并无实质性的关联，互不冲突。如果不想做到混淆可以取其他更有辨别的变量名，如 `add_result` 和 `sub_result`。

来交作业

编写一个函数来判断一个数是否为质数（素数）

题目描述：质数是大于 1 的自然数，且除了 1 和它本身以外，没有其他因数。你需要编写一个函数 `is_prime(n)` 来判断传入的整数 `n` 是否为质数，并返回布尔值 `True` 或 `False`。

函数定义：

```
def is_prime(n):
    pass
```

要求：

1. 输入一个正整数 `n`，并判断 `n` 是否为质数。
2. 如果 `n` 是质数，返回 `True`；否则返回 `False`。
3. 你需要在函数中使用循环和条件语句来检查 `n` 是否有其他因数。

输入输出示例：

```
print(is_prime(7)) # 输出: True
print(is_prime(10)) # 输出: False
print(is_prime(2)) # 输出: True
print(is_prime(1)) # 输出: False
```

3. 默认参数与可选参数

Python 函数支持为参数提供默认值，这使得某些参数在调用函数时可以省略。如果省略了参数，函数会使用定义时指定的默认值。

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("Alice")      # 输出: Hello, Alice!
greet("Bob", "Welcome") # 输出: Welcome, Bob!
```

在这里，`message` 是一个可选参数，它有一个默认值 `"Hello"`，如果在调用时不传入 `message` 参数，默认值会被使用。

4. 可变数量的参数

Python 支持传递可变数量的参数，使用 `*args` 表示任意数量的非关键字参数，`**kwargs` 表示任意数量的关键字参数。

```
def sum_numbers(*args):
    return sum(args)

result = sum_numbers(1, 2, 3, 4)
print(result) # 输出: 10
```

II.5. 数据抽象

数据抽象（Data Abstraction）数据抽象的核心思想是将“是什么”（What）与“怎么做”（How）分开。通过数据抽象，开发者只需要关心数据的使用方式，而不需要了解数据的内部实现细节。例如，在面向对象编程（OOP）中，数据抽象的一个常见例子是“类”。类定义了对象的属性（数据）和方法（行为），但隐藏了具体的实现细节。

II.5.1. 类与对象

类是 Python 实现数据抽象的主要工具。通过定义类，可以将数据（属性）和操作数据的方法（函数）封装在一起。在类的外部，用户无需了解类的内部实现，只需调用类提供的接口来操作数据。换句话说，类主要定义对象的结构，然后我们以类为模板创建对象。

II.5.1.1. 使用类

在 python 一切皆为对象，对象是类的实例。要使用类我们就要先对其进行实例化。这里我们使用 python 内置的 List 类进行演示。

```
l = list((3, 1, 2)) # 实例化 list 类，并生成对象 l
print(l)           # 输出对象 l 内的数据
# 输出 [3, 1, 2]
l.sort()          # 这是 List 类的内置函数，对 l 内的元素进行排序
print(l)
# 输出 [1, 2, 3]
```

从这个例子中我们可以看到 List 类的使用方法，它包含了数据和操作数据的方法。通过这样一个数据抽象的过程我们能更方便地处理问题。当然除了 List，在 python 中还有 Set, Dict, Tuple 等数据类型，后面我们会更加详细的介绍它们。

int 和 float

int 和 float 与数据抽象密切相关。尽管在 Python 中我们通常将它们视为基本数据类型，但它们实际上都是类，而这些类在设计中也运用了数据抽象的原则。

int 和 float 类提供了一组明确的接口（方法），通过这些接口，用户可以对这些对象进行各种操作，而不需要了解其背后的实现。例如：

```
a = 5
b = 2

# 你可以执行以下操作而不需要知道具体如何实现
print(a + b)      # 相加
print(a.__add__(b)) # int 类的内置加法方法，实际上相加操作就是由这个函数实现的
```

II.5.1.2. 创建类

当 python 内置的类已经不能满足你的需求了，你就可以自己进行数据抽象，创建一个类。下面是一个简单例子：

```
class Dog:  
    def __init__(self):      # 初始化函数  
        self.Race = 'dog'  
    def call(self):          # 方法  
        print('wang')  
  
a = Dog()  
a.call()      # 输出 wang  
print(a.Race) # 输出 dog
```

在这个例子中我们创建了一个狗的实例，它包含了最基础的数据 Race,还有方法 call。`__init__` 为一个特殊的方法，它会在类实例化时自动执行，也就是说在 a 被创建时，这个方法就自动执行了。`self` 通常用来指代现在的对象，它的存在使对象可以使用类中定义的方法和对象的数据。下面我们来进一步优化一下该类，给狗加一个名字，并加一个 play 的方法：

```
class Dog:  
    def __init__(self, name):  
        self.Race = 'dog'  
        self.name = name  
  
    def call(self):  
        print('wang')  
  
    def play(self):  
        self.call()  
        print(f'{self.name} 欢快的打滚')  
  
a = Dog('张致选') # 实例化  
a.play()          # 输出 wang 张致选欢快的打滚
```

这个例子中 `__init__` 处多加了一个参数 name，我们要在实例化时就输入该参数，因为 `__init__` 方法在实例化时就执行了。在赋值给 `self.name` 后我们就能在 `play()` 中使用它。（没有赋值将不能在别的函数中访问，这也是 `self` 的作用所在），同样在类的方法中我们也能互相调用，同样是通过 `self` 的方式引用。

类的使用还有更加复杂的方法，在后面的面对对象程序设计中，我们还会介绍更多的细节，包括继承和多态等，现在主要是要了解数据抽象的思想：把生活中复杂的物体抽象出来，对其进行封装成各种数据和方法，并方便我们的操作。

来交作业

抽象一个简单的图书馆

题目描述：建立一个图书馆类，存放历史，哲学，教育书籍。并能够统计整个图书馆的数据，能列出三种图书的数量，增加图书的数量

类的定义（参数未给全）：

```
class Library:  
    def __init__(self):  
        pass  
  
    def total(self):  
        pass  
  
    def list_book_num(self):  
        pass  
  
    def add_book(self):  
        pass
```

要求：

1. 要求包括数据： history ， philosophy ， educate ，分别表示历史，哲学，教育图书的数量。
2. 要求建立方法： total 计算整个图书馆图书数量并返回， list_book_num 列出每种图书的个数并返回字符串， add_book 能够增加这几种书的数量。

输入输出示例：

```
A = Library(100, 50, 60)  
print(A.total())  
A.add_book('his', 100)  
A.add_book('edu', 20)  
print(A.list_book_num())  
A.add_book('math', 200)  
  
# 输出  
# 170  
# History:200  
# Philosophy:50  
# Educate:80  
# There is no such book
```

III. 数据结构与标准库

III.1. 数据结构与 ADT

数据结构是指在计算机中存储和组织数据的方式。它是具体的实现，通常包括数据的存储格式和操作方法。思考一个问题，为什么我需要数据结构呢？如果从实际生活中看，想象一下你的房间里有很多东西，如果把所有东西都放在一个大箱子里，找东西时就会很麻烦。相反，如果你把书放在书架上，把衣服放在衣柜里，那么每次找东西就会方便多了。数据结构就是这样的“收纳方式”，帮助我们更高效地存储和访问数据。那么从计算机的角度来看，如果给你若干个数字，你会采取怎样的方式进行存储呢？学过变量之后，你可能这样想：

给定 n 个数,假定 n=3

```
a = 第一个数  
b = 第二个数  
c = 第三个数
```

但这样的方式极其不优雅，并且当 n 取值到很大的数时，我们也不可能手动去创建 n 个变量！因此是否能只使用一个变量去存储 n 个数，并且保证我们能快速访问到这任意 n 个数呢？回想一下，高中我们是否有使用一个符号代表一堆数的例子呢？自然地，我们使用 \mathbb{N} 表示自然数集合，我们使用一个符号表示某个集合，如 $a = \{1, 2, 3\}$ ，当然我们这里要讲的不是集合，在计算机世界里面，集合有其特殊性质，在这里我们只是希望找到一种简单存储数的方式。事实上，我们可以采用一种叫做数组(array)的数据结构去存储 n 个数，形式如下：

给定 n 个数

```
array = [第一个数,第二个数, ..., 第 n 个数]
```

我们可以通过下标的方式去访问这 n 个数

```
array[0] == 第一个数  
array[1] == 第二个数  
...  
array[n-1] == 第 n 个数
```

从 '0' 开始的计算机世界

在计算机世界中，绝大多数的数据结构的下标都是从 0 开始的，这样设置的原因在于计算机系统底层都是使用二进制存储，二进制中只有 0, 1 两种数字，因此当你向访问数组中第 1 个数字的时候，是使用 `array[0]` 的形式访问。当然如果你觉得别扭，我们以后统一从第 0 个数开始，`array[0]` 就是第 0 个数

通过上面数组这样的形式，理论上，给定任意 n 个数，都能够存储并且都能够快速访问。但事实上 Python 并没有提供叫做数组的数据结构，而是提供了一个叫做列表的数据结构 List(当然也是数组的一种，但是肯定是有特殊之处的)。如果要细分下来的话，数据结构有非常多，下面我们就依次介绍 Python 中的四大基本数据结构 List, Tuple, Dict, Set。

III.2. LIST

在 Python 中，`list` 是一种内置的数据结构，用于存储有序的元素集合。列表可以包含不同类型的元素，包括数字、字符串、甚至其他列表。以下是关于 Python 列表的一些常用操作：

1. 创建列表

可以使用方括号 `[]` 创建一个列表，也可以使用 `list()` 函数。

```
# 使用方括号创建列表  
my_list = [1, 2, 3, 'apple', 'banana']  
  
# 使用 list() 函数创建列表  
another_list = list((4, 5, 6)) # 从元组创建列表
```

试着创建一个列表

- » 创建一个 `my_list` 列表
- » 对列表元素进行遍历并打印

```
for i in my_list:  
    print(i)
```

注意

不同于 C 语言和 C++, Python 中 List 中的元素可以是任意类型，并没有规定同一个列表只能有一种类型，它可以包含任意的数据类型，float, int, 字符串等。当然在很久很久很久以后，出于可读性和安全的考虑，我们同样可通过某种声明来限定列表中的元素类型，其实是类型标注

2. 访问元素

列表的元素通过下标(索引)访问，索引从 0 开始。若从尾部开始则下标从 -1 开始

```
first_element = my_list[0] # 访问第一个元素，结果为 1  
last_element = my_list[-1] # 访问最后一个元素，结果为 'banana'
```

通过下标的方式访问列表元素

- » 通过下标的方式循环遍历列表元素
- ```
for i in range(len(my_list)):
 print(my_list[i])
```
- » 通过下标的方式逆序访问列表元素
- ```
for i in range(-1,-len(my_list),-1):  
    print(my_list[i])
```

3. 修改元素

可以通过下标(索引)修改列表中的元素。

```
my_list[1] = 'orange' # 将第二个元素修改为 'orange'
```

试着反转列表中的元素

如果列表元素为[1,2,3]，则反转后的列表元素应该为[3,2,1]

```
for i in range(len(my_list)//2):
    tmp = my_list[i]
    my_list[i] =
    my_list[-(i+1)] = tmp
```

4. 添加元素:

- » `append()`: 在列表末尾添加单个元素。
- » `extend()`: 在列表末尾批量添加元素。
- » `insert()`: 在指定位置插入元素。

```
my_list.append('grape') # 在末尾添加 'grape'
my_list.extend([1,2,3]) # 在末尾添加 1,2,3
my_list.insert(1, 'kiwi') # 在索引 1 处插入 'kiwi'
```

5. 删除元素:

- » `remove()`: 删除指定值的第一个匹配项。
- » `pop()`: 删除指定索引的元素，默认删除最后一个元素。

```
my_list.remove('apple') # 删除 'apple'
last_item = my_list.pop() # 删除最后一个元素并返回
```

6. 查找元素:

- » `index()`: 返回指定值的索引（如果不存在会引发异常）。

```
index_of_banana = my_list.index('banana')
```

7. 列表切片

可以使用切片操作符`:`获取列表的子集。用法为`list[起始下标:结束下标:步长]`,和`range`函数一样，左闭右开，默认步长为`1`。

```
sub_list = my_list[1:3] # 获取索引 1 到 2 的元素
sub_list2 = my_list[-3:] # 获取最后 3 个元素
sub_list3 = my_list[-3:-1] # 获取倒数第三到倒数第二的元素
sub_list4 = my_list[::-1] # 列表的逆序
sub_list5 = my_list[1:4:-1] # 索引 1 到 3 的元素的逆序
```

尝试不同的切片方式

我们看到了列表切片当中有三个可变参数，起始下标，结束下标，步长，上面我们已经给出了几种方式，动动你的手试一下吧

8. 列表推导式

Python 提供了一种简洁的方式来生成列表，称为列表推导式。这是一个语法上非常简洁的实现，当然你完全可以使用循环+分支的形式替代列表推导式，但是这样就代码就变得不那么优雅

```
squares = [x**2 for x in range(10)] # 生成 0 到 9 的平方数列表  
filter_squares = [x for x in squares if x%2==0] # 只保留 squares 中 2 的倍数
```

写点推导式吧

给定列表

```
my_list = list(range(21))
```

1. 使用列表推导式过滤掉列表中 3 的倍数的数
2. 使用列表推倒式将列表中所有数加 2

增删改查

我们可以看到 List 的操作有增加，删除，修改，查找元素(增删改查)这四种操作，事实上，绝大多数的数据结构的基本操作也都是增删改查，包括之后的 Tuple, Dict, Set 同样也是如此。所以当你学习某一个数据结构时，不知道从何入手，可以先关注这四个基本功能是如何实现的，再去关注不同数据结构之间的特性和差异，根据经验，按照这样的方式去理解和记忆，效率是比较高的。

III.3. TUPLE

在 Python 中，tuple（元组）是一种内置的数据结构，用于存储有序的元素集合。与列表相比，元组是不可变的，这意味着一旦创建，元组中的元素就不能被修改。以下是关于 Python 元组的一些关键点：

1. 创建元组

可以使用圆括号 () 创建元组，也可以使用 tuple() 函数。

```
# 使用圆括号创建元组  
my_tuple = (1, 2, 3, 'apple', 'banana')  
  
# 使用 tuple() 函数创建元组  
another_tuple = tuple((4, 5, 6)) # 从元组或其他可迭代对象创建
```

2. 访问元素

元组的元素通过索引访问，索引从 0 开始。

```
first_element = my_tuple[0] # 访问第一个元素，结果为 1  
last_element = my_tuple[-1] # 访问最后一个元素，结果为 'banana'
```

3. 修改元素不可变性：

修改元素？不，我们刚才讲了元组一旦创建，其元素无法更改，你不能添加、删除或修改元组中的元素，不然会报错，自己体验一下

```
# 尝试修改元素会引发错误  
# my_tuple[1] = 'orange' # TypeError: 'tuple' object does not support item assignment
```

4. 元组推导式

Python 不支持元组推导式，但可以通过生成器表达式和 `tuple()` 函数创建元组。

```
squared_tuple = tuple(x**2 for x in range(5)) # 结果: (0, 1, 4, 9, 16)
```

思考，动手

你可能会觉得元组好像没什么用？不能增加也不能修改元素，甚至连删除都做不到，跟列表功能那么丰富的数据结构没法比。但是既然存在就有其道理，其不变性正是我们需要的，因为在某些场景当中，我们要求数据必须是不可变，否则就会产生问题！

» 为了避免过于枯燥，下面你动手创建一个元组，并尝试修改元素内容

```
my_tuple = (1,2,3)  
my_tuple[0] = 1
```

» 试着将两个元组相加会产生什么结果呢？试着运行以下代码看看输出结果

```
my_tuple = (1,2)  
my_tuple2 = (3,4)  
print(my_tuple+my_tuple2)
```

» 将元组乘上一个数字会发生什么呢？

```
my_tuple = (1,2,3)  
print(my_tuple * 3)
```

» 其实 List 也支持相加和乘数的操作，你同样也可以去尝试下

III.4. DICT

字典，这个名词大家应该很熟悉了，计算机世界里面的字典和现实世界中的字典在功能上很相似。现实中，字典的一个字对应一块解释内容，计算机中，一个键对应一个值。在 Python 中，`dict`（字典）是一种内置的数据结构，用于存储键值对。字典是无序的、可变的，并且可以使用不可变的类型（如字符串、数字、元组）作为键。以下是关于 Python 字典的一些关键点：

1. 创建字典

可以使用花括号 `{}` 或 `dict()` 函数创建字典。

```
# 使用花括号创建字典  
my_dict = {  
    'name': 'Alice',  
    'age': 30,  
    'city': 'New York'  
}  
  
# 使用 dict() 函数创建字典  
another_dict = dict(name='Bob', age=25, city='Los Angeles')
```

2. 增加元素

直接通过新的键赋值来添加。

```
my_dict['email'] = 'alice@example.com'
```

3. 删除元素 可以使用 `del` 语句或 `pop()` 方法

```
# 删除元素  
del my_dict['city'] # 使用 del  
age = my_dict.pop('age') # 使用 pop() 返回被删除的值
```

4. 访问和修改元素

可以通过键访问字典中的值，并且可以直接修改值。

```
# 访问元素  
name = my_dict['name'] # 结果: 'Alice'  
# 修改元素  
my_dict['age'] = 31 # 将年龄修改为 31
```

5. 常用方法

- » `keys()`：返回字典中所有键的视图。
- » `values()`：返回字典中所有值的视图。
- » `items()`：返回字典中所有键值对的视图。

```
keys = my_dict.keys() # 获取所有键  
values = my_dict.values() # 获取所有值  
items = my_dict.items() # 获取所有键值对
```

试着遍历下

看下输出结果，你就知道这些函数到底是干什么用的了

```
for key in my_dict.keys():  
    print(key)  
    print("====")  
for value in my_dict.values():  
    print(value)  
    print("====")  
for item in my_dict.items():  
    print(item)
```

6. 字典推导式

Python 支持字典推导式，使得创建字典更加简洁。

```
squared_dict = {x: x**2 for x in range(5)} # 结果: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

写点代码

使用字典统计列表中的每个元素出现的次数如 `my_list = [1,1,2,2,3]` 则 `my_dict={1:2,2:2,3:1}`

» 提示：你可以使用 `in` 来判断某个元素是否在字典当中

III.5. SET

这里的集合和数学上的集合可以认为是等价。在 Python 中，`set`（集合）是一种内置的数据结构，用于存储唯一的、无序的元素集合。集合提供了高效的成员测试和集合运算。以下是关于 Python 集合的一些关键点：

1. 创建集合

可以使用花括号 `{}` 或 `set()` 函数创建集合。

```
# 使用花括号创建集合
my_set = {1, 2, 3, 'apple', 'banana'}

# 使用 set() 函数创建集合
another_set = set([4, 5, 6]) # 从列表创建集合
```

2. 集合特性

» 无序性：集合中的元素是无序的，因此不能通过索引访问。
» 唯一性：集合中的元素必须是唯一的，如果重复添加，集合会自动去重。也就是集合元素不能重复

```
duplicate_set = {1, 2, 2, 3}
# 结果: {1, 2, 3}
```

3. 访问集合元素

由于集合是无序的，不能通过索引访问，但可以使用 `in` 关键字检查某个元素是否存在。

```
if 'apple' in my_set:
    print("苹果在集合中")
```

4. 集合操作

» 添加元素：使用 `add()` 方法添加单个元素。

```
my_set.add('orange') # 添加 'orange' 到集合
```

» 删除元素：使用 `remove()` 或 `discard()` 方法删除元素。

```
my_set.remove('banana') # 删除 'banana'，如果不存在则抛出异常
my_set.discard('grape') # 删除 'grape'，如果不存在则不抛出异常
```

» 清空集合：使用 `clear()` 方法清空集合。

```
my_set.clear() # 清空集合
```

5. 集合运算

集合支持多种运算，包括并集、交集、差集和对称差集。这些集合运算和数学上的集合是一致的，并不想做过多解释，请 STFW 自行了解下吧

» 并集：使用 `union()` 或 `|` 运算符。

```
set1 = {1, 2, 3}  
set2 = {3, 4, 5}  
union_set = set1 | set2 # 结果: {1, 2, 3, 4, 5}
```

» 交集：使用 `intersection()` 或 `&` 运算符。

```
intersection_set = set1 & set2  
# 结果: {3}
```

» 差集：使用 `difference()` 或 `-` 运算符。

```
difference_set = set1 - set2  
# 结果: {1, 2} 集合 1 有的元素而集合 2 没有的元素
```

» 对称差集：使用 `symmetric_difference()` 或 `^` 运算符。

```
symmetric_difference_set = set1 ^ set2  
# 结果: {1, 2, 4, 5} 两个差集的并集
```

6. 集合推导式

Python 支持集合推导式，可以快速创建集合。

```
squared_set = {x**2 for x in range(5)}  
# 结果: {0, 1, 4, 9, 16}
```

写点代码

这一部分内容确实无趣，但是确实是需要经历的过程，有时候很难保证学习的过程是有趣的。因此来写点代码吧

» 给定两个列表，求两个列表的共同元素

关于数据结构

到这里我们就介绍完了 python 中基本的数据结构，还有更多复杂的数据结构我们么有介绍，可能在很久以后我们会特别出一个章节来讲数据结构。其实其它编程语言大抵上也会提供类似 List,Dict,Set 这样的数据结构，掌握好每种数据结构的特点和基本的增删改查，学习其他编程语言的门槛也就不会那么高了。

III.6. ALGORITHMS

介绍完数据结构，现在我们来介绍下一些基本算法。算法可以通俗地理解为解决问题的“步骤说明”或“食谱”。就像做饭时需要按照食谱的步骤来准备和烹饪食材，算法则是告诉我们如何通过一系列明确的步骤来完成某项任务或解决某个问题。

1. 明确性：每一步都必须清楚明了，不能有任何模糊的地方。就像食谱中的每个步骤都要具体，比如“切洋葱”而不是“处理洋葱”。
2. 有序性：步骤需要按照特定的顺序进行。有些步骤是前置的，必须在后续步骤之前完成。比如在煮汤之前，必须先把食材准备好。
3. 有限性：算法应该在有限的步骤内完成，也就是说，当你按照步骤执行后，最终会得到一个结果，而不是无限循环。
4. 输入与输出：算法通常会接收输入（如食材），经过处理后产生输出（如熟食）。输入可以是以任何形式的数据，而输出是最终结果。

举个例子：假设你要把一堆数字从小到大排序。这个问题的算法可能包括以下步骤：

-
1. 从头到尾查看每个数字。
 2. 找到当前未排序部分中最小的数字。
 3. 把这个最小的数字放到已排序部分的末尾。
 4. 重复步骤 1 到 3，直到所有数字都排序完成。
-

在这个例子中，你可以看到每一步都是明确的（算法步骤），经过有限次的循环后结束程序（有限性），并且有一个清晰的开始和结束（结束点是这堆数据已经有序），输入是一堆数字，而输出是这堆数字的有序排列（输入与输出）。简而言之，算法就是解决问题的步骤和方法，可以用于计算、数据处理、搜索、排序等多个方面。掌握算法的思维方式有助于我们更有效地解决各种问题。

建议

在解释某个名词的过程，我们希望能有一些专业性的解释，难免会有一些类似八股文的内容，不是要求大家死记硬背，而是理解性的记忆，并且尽可能思考其中提出这样理论的动机。就比如当你拿到一份代码的时候，你能够快速分析，得到程序的输入输出，以及每一步算法大概的作用，能够从你的理解出发而系统性地解释一份代码，这些需要一点系统和理论的指导，当然如果你觉得这是没有必要过程，可以跳过，这种能力是需要通过阅读大量代码才能慢慢培养而成的。

上面介绍的例子其实就是选择排序的算法过程，为了便于理解，此处再举几个例子：

- » 当我们要求两个数之和时，其中的 加法 就是一个算法
- » 如果我们要求 1-100 的数字之和，那么这个问题的解法也是一个算法
- » 如果我们要求一堆数中的最大数，那么求出这个最大数的过程是一个算法

现在你可以理解“算法是解决问题的方法或过程”这句话的含义了吧。接着我们介绍 python 中内置的一些基本算法

III.6.1. 排序

什么是排序呢？排序就是将无序的序列变成有序的，对于数字来讲，有从小到大的排序，有从大到小的排序。前面我们已经提到了一个选择排序的例子，那么你能够根据算法原理而给出代码实现呢？下面是选择排序的一种实现方式

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        # 假设当前索引 i 是最小值的索引
        min_index = i
        # 在未排序部分寻找最小值
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # 交换找到的最小值和当前索引 i 的值
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

my_list = [64, 25, 12, 22, 11]
sorted_list = selection_sort(my_list)
print("排序后的列表:", sorted_list) # 输出: [11, 12, 22, 25, 64]
```

看不懂代码？

看不懂是正常的，我们并没有特别详细地讲解选择排序的原理，因为本手册的重点并不在于教大家算法，那这个时候怎么办呢？STFW

当然如果每次需要数进行排序，都要写一遍这样的函数多麻烦，其实 Python 中内置了 `sorted()` 函数可以对可迭代对象进行排序，其内部实现是一种叫做 Timsort 的排序算法，感兴趣可以自行去了解下。下面是 `sorted` 函数的使用案例

```
# 对列表进行排序
my_list = [5, 2, 9, 1, 5, 6]
sorted_list = sorted(my_list) # 默认升序,reverse=False
print(sorted_list) # 输出: [1, 2, 5, 5, 6, 9]
# 使用 reverse 参数进行降序排序
sorted_list_desc = sorted(my_list, reverse=True)
print(sorted_list_desc) # 输出: [9, 6, 5, 5, 2, 1]
```

可迭代对象

简单的理解就是，内部的元素是有序的数据结构，你可以遍历里面的元素

遍历就是挨个访问数据结构内的元素

III.6.2. 查找最大值和最小值

使用内置的 `max()` 和 `min()` 函数可以快速找到可迭代对象中的最大值和最小值。

```
# 查找最大值和最小值
max_value = max(my_list) # 结果: 9
min_value = min(my_list) # 结果: 1
```

试着实现下

» 不用 `max` 函数，你能否实现从一个列表中找出最大值呢？下面这是一种简单实现呢？

```
def max_num(my_list):
    max_n = my_list[0]
    for num in my_list:
        if(num > max_n):
            max_n = num
    return max_n

my_list = [342,234,564,23,456]
print(max_num(my_list))
```

» 那么实现从列表中找出最小值，你应该也会了吧？

» 现在我希望从列表中找出第二小的值，又该如何实现呢？动手尝试一下吧

III.6.3. 求和与平均值

可以使用 `sum()` 来计算总和，结合 `len()` 计算平均值。也可以直接使用 `mean()` 计算平均值

```
# 计算总和和平均值
total = sum(my_list) # 结果: 28
average = total / len(my_list) # 结果: 4.67

# 直接计算平均值
average = mean(my_list)
```

III.6.4. 过滤和映射

使用 `filter()` 和 `map()` 函数可以对可迭代对象进行过滤和映射操作。

- » `filter(func, iterable)`，第一个参数为函数，第二个参数为可迭代对象。通常我们喜欢使用 `lambda` 函数当作第一个参数，第二个参数可以是列表。功能是只保留可迭代对象中的满足 `func` 的元素。如果你觉得上面的表述太难懂了，可以跳过，关注于如何使用这个函数即可。
- » `map(func, iterable)`，参数和 `filter` 是一样的，功能是对可迭代对象中的每个元素做变化，如做平方等。

```
# 过滤,只保留偶数
even_numbers = list(filter(lambda x: x % 2 == 0, my_list)) # 结果: [2, 6]

# 将每个元素平方
squared_numbers = list(map(lambda x: x ** 2, my_list)) # 结果: [25, 4, 81, 1, 25, 36]
```

熟悉起来

给定一个列表，使用 filter 和 map 完成下列任务

- » 过滤列表中类型不为 int 的元素
- » 对列表中的每个元素取绝对值，可以使用 abs() 函数对数取绝对值

匿名函数和高阶函数

关于 lambda，其实叫做匿名函数，而 filter 和 map 这样可以接受函数为参数的函数，我们称之为高阶函数。这些概念我们会在后面的函数式编程中为大家作解释。

III.6.5. 集合操作

事实上，集合操作也是一种算法，前面介绍集合的时候已经给出，这里不再赘述

III.6.6. 字符串操作

前面我们说到了字符串，在 python 中字符串的操作是比较灵活的，其实字符串具备和列表一样的切片功能

```
s = "hello,world!"  
sub_s = s[0:5:1]  
print(sub_s)
```

Python 也提供了丰富的字符串方法，如 join()、split()、replace() 等。

- » join(word)，将可迭代对象(列表，元组等) words 使用调用该函数的字符串拼接起来，返回一个字符串
- » split(word)，以 word 为间隔符，将字符串分割开，返回一个列表
- » replace(word,new_word)，将字符串中 word 全部替换成 new_word，返回替换完的字符串

```
# 字符串连接  
words = ['Hello', 'World', '!']  
sentence = ''.join(words) # 结果: 'Hello World !'  
  
# 字符串分割  
split_sentence = sentence.split(' ') # 结果: ['Hello', 'World', '!']  
  
# 替换字符串  
new_sentence = sentence.replace('World', 'Python') # 结果: 'Hello Python !'
```

试着实现这几种函数呢

» join 的内部实现你是否能够大概感知到呢？下面是一种实现方式

```
def join(seperator,words):  
    s = ""  
    for word in words[:-1]:  
        s = s + word + seperator  
    s += words[-1]  
    return s  
  
seperator = ''  
words = ["hello","world","!"]  
s = join(seperator,words)  
print(s)
```

» 试着动手实现下 split 和 repalce 函数吧

数据结构与算法

关于数据结构与算法，我们在这里只是非常非常简单地介绍了一些数据结构与算法，如果你想了解更多，可以从 [hello algo!](#) 看起，对大家之后的程序设计课有很大帮助，事实上，当系统学完该手册后，你也应该去学数据结构与算法了

到这里，我们的第三部分也结束了。接下来我们会介绍一些编程语言理论相关的知识，这些都是宝贵的编程思想。

III.7. 模块

模块是 Python 中的一种代码组织方式，可以把相关的代码放在一个文件里，方便管理和重用。通俗地说，模块就像是一个工具箱，你可以把常用的工具（函数、类、变量等）放在里面，其他地方想用的时候只需要打开这个工具箱就可以了。

1. 代码分组：

» 模块允许你将相关的功能放在一个文件中，比如把所有与数学计算相关的函数放在一个 `math_utils.py` 的文件中。

2. 重用性：

» 一旦创建了模块，你可以在其他程序中导入它，避免重复编写相同的代码。

3. 命名空间：

» 每个模块都有自己的命名空间，这意味着模块内部定义的变量和函数不会与其他模块的变量和函数冲突。

举个例子 想象你在做一个项目，需要一些常用的计算功能，比如加法、减法等。你可以创建一个模块

`calculator.py` :

```
# calculator.py
```

```
def add(x, y):
```

```
return x + y
```

```
def subtract(x, y):  
    return x - y
```

然后在你的主程序中，你可以使用 `import` 来导入这个模块，使用里面的函数：

main.py :

```
# main.py  
  
import calculator  
  
result1 = calculator.add(5, 3)  
result2 = calculator.subtract(10, 4)  
  
print(result1) # 输出: 8  
print(result2) # 输出: 6
```

上面这个的例子的目录结构应该是

- 项目文件夹
- main.py
- calculator.py

Python 中有很多这样的内置模块，这些内置模块也成为标准库，利用这些已有的模块，不需要自己再去实现复杂的函数，就可以快速完成程序的编写。下面介绍一些常见的内置模块吧

III.7.1. MATH

主要提供一些数学运算的函数，直接看具体的例子吧，主要还是学会如何使用这些函数

```
import math # 导入 math 模块(库)  
  
print(math.pi) # 圆周率  
print(math.sqrt(4)) # 开根号  
print(math.log(2)) # 对数函数  
print(math.sin(2)) # 三角函数  
...
```

还有其他很多很多的函数在 `math` 模块里面，请自己去尝试一下吧。

III.7.2. TIME

时间模块，提供一些时间相关的函数

```
import time  
  
# 获取当前时间戳
```

```
current_time = time.time()
print(current_time)

# 暂停执行 2 秒
time.sleep(2)
print("Waited for 2 seconds.")
```

程序计时

- » 使用 time 模块完成对某一个程序的运行时间计时
- » 提示：用程序最后的时间点减去程序开始的时间点

III.7.3. DATETIME

日期模块，可以获取系统的日期，还有一些其他时间相关的函数

```
from datetime import datetime

# 获取当前日期和时间
now = datetime.now()
print(now)

# 格式化日期
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

III.7.4. RANDOM

随机库，使用一些随机函数，如取随机值，随机抽样等

```
import random

# 生成一个随机浮点数
print(random.random())

# 从列表中随机选择一个元素
fruits = ['apple', 'banana', 'cherry']
print(random.choice(fruits))
```

随机取数字

- » 请 STFW 如何使用 random 完成，在某一区间内取随机数

III.7.5. OS

用于操作操作系统功能，如文件和目录操作。

```
import os

# 获取当前工作目录
print(os.getcwd())

# 列出当前目录下的文件
print(os.listdir('.'))
```

更多的模块

Python 中还有很多内置模块，如 json, csv, sqlite, request 等，请自行 STFW 学习吧

当然你还可以下载第三方模块（就是需要下载和安装的），Python 目前有丰富的第三方库，你可以通过 pip 安装，具体的请看工具链那一部分的包管理工具

IV. 编程范式入门

IV.1. 面向过程编程

面向过程编程（Procedural Programming）是一种编程范式，它将程序视为一系列按顺序执行的步骤或过程。这个范式强调通过过程（或函数）来组织代码，通常包括以下几个特点：

1. 模块化：程序被分解为多个小的、可重用的过程或函数，每个过程执行特定的任务。
2. 顺序执行：程序的执行是线性的，通常是从上到下逐行执行。
3. 变量和数据：通过使用变量来存储数据，并在过程之间传递数据。
4. 控制结构：使用控制结构（如条件语句和循环）来控制程序的执行流。

面向过程编程的核心是过程（函数），使用这种方式设计程序时，通常采用自顶向下的开发方式，先确定程序的最终输出，再通过将功能拆分为多个子功能，并逐级向下，直到将所有功能都拆分成比较原始函数为止。

考虑一下这样的场景：需要设计一个简易通讯录，该通讯录具备，添加联系人，删除联系人，查看联系人的功能，你会怎样设计呢，事实上你很可能这样设计

```
def 添加联系人(联系人信息)
def 删除联系人(联系人信息)
def 查看联系人()

def 初始化通讯录()
```

我们将通讯录的功能拆分成几个函数去实现，分成几个模块完成对应的功能，事实上这几个功能函数可能还可以进行拆分，比如添加联系人这个操作，我们还可以设定一些判定操作，比如，判定输入的电话号码是否合法，给这个联系人添加头像等等之类更复杂和细的操作。就是通过这种功能模块过程的逐渐划分，最终我们可以得到一套设计方案，这就是所谓的面向过程。

IV.2. 面向对象编程

注意

- » 前面我们已经讲述过数据抽象的内容了，如果你觉得你已经掌握了数据抽象，可以跳过这部分的引言，直接关注面向对象的三大特性。
- » 本章的内容会有大量的知识点（一些基础的抽象思想），有些内容是需要反复理解，如果你觉得这个过程痛苦，不妨自己从现实的角度去考虑计算机世界里的这些抽象问题的必要性，我想说的是，我们只提供一个角度的思考和学习方式（能力和写作水平有限），你要有自己的思考和想法。学习本身应该是自由的

面向对象编程（OOP Oriented Object Programming）可以通俗地理解为一种编程方式，它把现实世界中的事物抽象成“对象”，并通过这些对象来组织和管理代码。那么什么是对象呢？想象一下，你在生活中看到的各种事物，比如“猫”、“汽车”。在编程中，这些事物就被称为对象。每个对象都有自己的特点（属性）和可以做的事情（方法）。比如，猫的属性可能是“颜色”和“年龄”，而它的方法（行为）可能是“猫叫”和“吃”。

为什么需要对象这个概念呢？试着想一下，如果要使用编程语言描述通讯录中的联系人呢，你会怎么做？首先我们肯定会想，一个联系人有姓名，电话，年龄，性别等，假设有一人叫做张三，电话 18800000000，年龄 23，性别男，自然地可以像下面的代码这样描述

```
name = "张三"  
age = 23  
sex = "男"  
tel = "18800000000"
```

但是这样的表示实在是不优雅呀，如果再来一个叫李四的人，你又得去手动创建 4 个变量来存储李四的信息。那么有没有办法简化这一过程呢？

既然张三，李四，在属性上形式都一样，只是具体的取值不同，此时我们就可以将其抽象成类(class)，类可以理解为一个类别，比如动物，植物是一种类别，通过类来创建对象。在 Python 中，我们可以使用关键字 class 来对某一个类别进行声明，具体的形式如下：

```
class Person:  
    name = 姓名  
    age = 年龄  
    sex = 性别  
    tel = 电话
```

那么既然已经将人这个类抽象出来了，如何通过这个类来创建对象呢？事实上，Python 中的 class 都自带一个 `_init_()` 的函数(方法)（即使你不去声明，它本身仍存在一个无参数的 `_init_` 方法），用于提供初始化对象（如其函数名 `initialize`），我们可以通过这个方法来初始化对象的一些信息，但是其实我们不用显式的调用它，而是像使用函数一样使用 `类名(参数列表)` 的方式去创建对象即可，具体实现如下：

```
class Person:  
    def __init__(self, name, age, sex, tel):  
        self.name = name  
        self.age = age  
        self.sex = sex  
        self.tel = tel  
  
Person1 = Person("张三", 23, "男", "18800000000") # 调用 __init__ 函数创建一个对象  
Person2 = Person("李四", 18, "男", "18810000000")
```

self

`self` 其实就是对象本身，这么讲可能有点难懂。其实在调用 `__init__` 函数时，已经创建了一个对象，这个对象在类内就是 `self`，然后通过 `self.属性` 的方式对属性的进行初始化和赋值

```
调用 ##### 产生 赋值 #####  
Person()----> # __init__ # ----> self ----> # self.属性 = 传入的属性 #  
#####
```

上面的过程讲述了如何使用 Python 的 class 声明类，和使用类来创建对象，那么创建了对象之后，如何访问对象对应的属性值呢？使用 . 就可以访问一个对象的属性了

```
print(Person1.name) # 张三  
print(Person1.age) # 23  
print(Person1.sex) # 男  
print(Person1.tel) # 18800000000
```

要更改对象属性值也很简单，直接通过赋值的方式

```
Person1.age = 18  
print(Person1.age) # 18
```

动动手指

- » 创建一个商品类，有商品名称，价格，生产地，商家
- » 创建一个商品类对象
- » 修改商品类对象的任意属性

前面提到了如何声明类和创建类对象，但这样就结束了吗？那这个结构也太简单了。事实上，一个“类别”总有某些特殊的行为，就像人会走路，吃饭，睡觉，鸟会飞，鱼会游一样，这些行为在类里面叫做方法。我们可以在类里定义函数，类里面的函数就是方法了，具体可以看下面几个例子

```
# 人  
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def eat():  
        pass  
    def sleep():  
        pass
```

```
# 鸟  
class Bird:  
    def __init__(self, age):  
        self.age = age  
    def fly():  
        print("flying")
```

我们同样可以使用 对象.方法 的形式使用类的方法

```
bird1 = Bird(18)  
bird1.fly() # flying
```

__init__

为什么我说，即使不去声明 `__init__` 它本身也有一个 `__init__` 方法呢？事实上，我们可以认为在 Python 中所有类型皆是对象，而这些类都有一个共同祖先 `Object`，`Object` 具有 `__init__` 这个方法（无参数的），Python 中所有类都会继承 `Object`，因此具备 `Object` 这个类的所有属性和方法。至于什么是继承，请往下看

IV.2.1. 面向对象三大特性

面向对象中有三大特性，分别是封装，继承，多态。接下来我们会举一些简单的例子帮助大家理解

IV.2.1.1. 封装 ENCAPSULATION

想象一下这样的场景：如果你有一个银行账户，你肯定不希望你的账户信息能被随意访问或者篡改，因此我们必须约束外界对我们账务信息的访问。在面向对象中，这样的形式我们称之为封装。所谓封装，就是把数据和操作这些数据的方法放在一起，并保护这些数据不被随意修改。就像一个黑盒子，你只能通过盒子外面的按钮来控制盒子内部的机制，而不能直接看到或改动里面的东西。这种机制能够保护对象的内部状态，防止外部代码直接访问和修改，从而提高了系统的安全性和可维护性。

以下是一个简单的 Python 示例，演示了封装的概念：

```
class BankAccount:
    def __init__(self, balance=0):
        self.__balance = balance # 私有属性，外部无法直接访问

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrawn: {amount}")
        else:
            print("Invalid withdraw amount.")

    def get_balance(self):
        return self.__balance # 提供公共方法访问私有属性

# 使用封装
account = BankAccount()
account.deposit(100)      # 输出: Deposited: 100
account.withdraw(50)      # 输出: Withdrawn: 50
print(account.get_balance()) # 输出: 50
```

我们来解释一下上面的代码:

首先我们定义了一个 BankAccount 类，为其设定一个私有属性 `_balance`（什么是私有属性？请往后看看），外部代码无法直接访问。由于私有方法是不能直接通过外界访问的，因此我们还定义了 `deposit`、`withdraw` 和 `get_balance` 这些公共方法，提供了对私有数据的安全访问和操作。

接着创建了一个 `account` 对象，使用 `deposit` 方法存 100 进 `account` 的账户中，使用 `withdraw` 从 `account` 对象中取出 50，通过 `get_balance()` 获取 `account` 对象中的账户余额。

通过这样的方式，我们封装了 `BankAccount` 的内部实现，外界程序不能直接修改和访问其私有程序，并且我们向外界提供了公共方法实现 `BankAccount` 的一些操作，当然这些操作外界也不知道具体是什么，只能接收到一个结果。这样极大程度上保证了 `BankAccount` 的内部安全，同时只对外提供必要的接口(方法)，也能简化程序的实现

私有属性和私有方法

私有属性是指那些不能被外部直接访问的属性。在 Python 中，在类中使用 `_` 开头的属性就是私有属性。那么具体是什么意思呢？看下面这个例子

```
class Person:  
    def __init__(name,tel):  
        self.name = name  
        self.__tel = tel  
  
P = Person("张三","18800000000")  
print(P.name)  
print(P.__tel)
```

如果你运行上面这个代码你会发现 `print(P.__tel)` 这一行运行时会报错，原因在于与 `__tel` 在 `Person` 类中是私有的，你不能在 `Person` 类外直接访问，只能在类内部使用，类的内部即为，声明类的整个过程，从 `class` 到缩进结束。

所谓的私有方法也是类似的，在 Python 中，类中以 `_` 为开头的方法即为私有方法

```
class Person:  
    def __private():  
        printf("private")  
    def func(self):  
        self.__private()  
  
P = Person()  
P.func()  
P.__private()
```

运行上面代码你会发现 `P.__private()` 这一行报错，因为 `__private` 是私有方法，但是你可以通过定义 `func` 函数的方式间接使用 `__private` 方法

任务

- » 为之前实现的商品类添加私有属性，销量，库存
- » 为商品类添加私有方法：出售商品(想想出售一个商品的时候会发生什么行为呢?)

IV.2.1.2. 继承 INHERITANCE

想象一下这样的场景吧，如果你已经创建了动物这样的类别，它已经具备动物类里面的基本行为，奔跑，进食，睡眠等行为，假设你也已经实现了这样的方法。那么如果此时又需要你创建一个狗类呢？你是不是也要重新实现，奔跑，进食，睡眠的方法，如果这样的话，岂不是很麻烦，最好的方法是让狗类能直接拥有动物类的行为和属性，那要怎么实现呢？答案是使用继承。

所谓继承是一种让新类可以从已有类获取属性和方法的机制。允许一个类（子类）基于另一个类（父类）创建新类，从而实现代码的重用和扩展。通过继承，子类具备了父类的所有的属性和方法，并可以重写或实现这些方法，以满足特定需求。比如，Animal类可以是一个更一般的类，而Cat和Dog可以从Animal继承一些共同的特性。具体地可以看下面这个例子

```
class Animal:  
    def __init__(name,age):  
        self.name = name  
        self.age = age  
    def speak(self):  
        print("叫")  
    def run(self):  
        print("跑!")  
  
# 声明狗类  
class Dog(Animal):  
    def speak(self):  
        print("汪汪汪!")  
  
# 声明猫类  
class Cat(Animal):  
    def speak(self):  
        print("喵喵喵!")
```

我们来解释一下上面这段代码

首先我们声明了Animal这个父类(基类)，定义了__init__和speak方法，

然后我们声明了Dog类，继承了Animal的属性和方法speak和run，但是又重新声明了speak的实现，这种子类重新声明父类的方法形式我们称为方法重写，如果不对方法重写，那么子类调用方法时就是使用父类的方法，比如run

然后又声明了Cat类，类似于Dog重新声明了speak方法，但是可以看到实现的内容和Dog的speak是不同，这也正常嘛，因为和的叫声不同

子类对象可以调用父类已经实现的方法。我们可以创建一个实例 dog，调用 run，然后调用 speak。同样的可以创建 cat

```
dog = Dog("小七",3)
dog.run() # 跑
dog.speak() # 汪汪汪
```

```
cat = Cat("小黑",2)
cat.run() # 跑
cat.speak() # 喵喵喵
```

任务

- » 新建一个狼类，继承 Animal，狼类有呼啸，奔跑等行为
- » 在这样一个场景：在汽车城里，有很多种类的车，燃油车，电车，甚至两轮电动车。车具备的行为是行驶，但是消耗的燃料不一样。请你设计这几个类的实现把

IV.2.1.3. 多态 POLYMORPHISM

多态允许不同类型的对象以相同的方式使用相同的方法。比如，不管是猫还是狗，它们都有一个“叫”的方法，但具体的叫声可能不同，也就是具体的函数内容是不同的。你可以用相同的方式调用它们的“叫”方法，这就是多态。多态有两种，分为运行时多态和编译时多态

1. 编译时多态（静态多态）：

- » 通过方法重载（同一类中定义多个同名但参数不同的方法）和运算符重载实现。
- » 例如：在 Python 中，虽然不支持方法重载，但可以通过默认参数或可变参数实现类似效果。

具体是什么意思呢？看下面这个例子

```
class Animal:
    def eat(self, food=" "):
        print("吃" + food)
A = Animal()
A.eat() # 吃
A.eat("草") # 吃草
```

我们来解释一下上面这个代码

首先，我们定义了 Animal 这个类，声明了 eat 这个方法

接着我们创建了对象 A，调用两次 eat 方法，两次传入不同的参数，产生了不同结果，这种在我们编写 eat 方法时就可以预见的不同结果的方式就是编译时多态，即根据函数内部实现就能推测出最后的几种执行结果。

方法重载

所谓方法重载就是，方法名相同，但是参数列表不同，如下面的两个方法，方法名相同，但参数的个数不同。

```
def add(a,b):  
    def add(a,b,c)
```

但是实际上，在 Python 中，不允许在同一个类中定义两个同名的方法，因此我们只能通过可变参数的方法，实现类似与方法重载的功能。

运算符号重载也是类似的，只不过是对 `__add__` 这中内置的运算符函数进行重载，具体的案例 STFW 吧

2. 运行时多态

- » 通过继承和方法重写（子类重写父类的方法）实现。这些我们前面提到过
- » 具体调用哪个方法在运行时决定，这种多态性是面向对象程序设计中最常见的形式。

具体什么是运行时多态呢？请看下面这个例子，

```
class Animal:  
    def speak(self):  
        raise NotImplementedError("Subclasses must implement this method")  
  
class Dog(Animal):  
    def speak(self):  
        return "Woof!"  
  
class Cat(Animal):  
    def speak(self):  
        return "Meow!"  
  
def make_animal_speak(animal):  
    print(animal.speak())  
  
# 使用多态  
dog = Dog()  
cat = Cat()  
make_animal_speak(dog) # 输出: Woof!  
make_animal_speak(cat) # 输出: Meow!
```

我们来解释一下上面这段代码：

`Animal` 是一个基类，定义了一个抽象方法 `speak()`，它的子类 `Dog` 和 `Cat` 实现了这个方法。

多态的实现：`make_animal_speak` 函数接受一个 `Animal` 类型的参数，并调用其 `speak()` 方法。根据传入对象的不同，具体调用的 `speak()` 方法也不同。

上面这样的形式就多态的一种体现，事实上在我们定义 `make_animal_speak` 时，我们根本不知道传入的参数 `animal` 是一个什么类型，只有在真正传入对象后，运行的过程才能确定其是 `Dog` 类还是 `Cat` 类，从而去调用相应类的方法。

抽象方法

抽象类，为何称其为抽象，原因在于与抽象类不适合实例化（创建对象），就比如你说水杯是一个物品，苹果是一个物品，手机也是一个物品，如果你要使用物品这个类创建对象，你认为这是否合适呢？过于高度的抽象本身并不适合在程序中使用，但你可以通过子类继承抽象类的方式实现抽象类的方法，进而创建子类对象

在面向对象里，抽象方法通常是定义在抽象类里，一般来说，抽象类不能创建对象，因此也不能直接调用抽象方法，必须通过子类继承抽象类，子类重写抽象方法，调用子类的方法的形式来实现。具体的例子就是上面提到的 `Dog` 实现了父类 `Animal` 的 `speak` 抽象方法

再来设计通讯录

- » 在讲述面向过程编程中，我们举了一个通讯录的例子，那么现在你能否使用面向对象的思想设计一个通讯录呢？
- » 坦克大战！如果你玩过坦克大战的话（4399 里的，时代的眼泪啊……）我们都应该知道，坦克的行为有，转向，前进，射击等，现在让你设计，你应该也会吧？当然坦克大战里面还有，子弹，地图，围墙等对象呢？你能否设计出原型呢？尝试思考或者写点代码吧。

游戏

你好奇游戏是怎么制作而成的吗？事实上，游戏就是通过编写程序而成，里面的各种对象的行为都是通过程序定义而执行，而你所看到的精美画面，只不过是通过计算机(cpu,gpu)背后计算后渲染而成的。因为游戏背后也是严密的逻辑，一个行为同样可以拆分成好几个动作和对应的画面最后呈现到大家面前。如果你想去做一个自己的游戏，请自己去深入学习编程的抽象思维吧

IV.3. 函数式编程

函数的话，我们都知道，那么函数式编程又是个什么东西呢？所谓的函数式编程就是，主要关注使用“函数”来处理数据，而不是通过改变状态或使用命令来控制程序的行为。它就像数学中的函数，给定相同的输入总会得到相同的输出。函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！这么讲还是太抽象了（），简单来说，函数式编程就是要求你尽量将运算过程写成一系列的函数调用，具体来看一下下面的这个例子吧

通常，如果要求我们完成下面这个运算 $(1+2)*3-4$ 的话，一般的做法是

```
a = 1 + 2  
b = a * 3  
c = b - 4
```

如果使用函数式编程的话，写法是这样的

```
def add(a,b):
    return a + b
def mul(a,b)
    return a * b
def sub(a,b)
    return a - b

res = sub(mul(add(1,2),3),4)
```

这就是函数式编程

至于为什么发明这样的写法（这看起来可复杂多了），那原因可太多了，现在告诉你们大概是无法理解（笔者也只有粗浅的认知）。事实上，上面的程序写成下面这样就好看很多了

```
add(1,2).mul(3).sub(4)
```

可以看到执行的顺序一目了然，先执行 `add` 再执行 `mul` 然后执行 `sub`。下面我们来正式介绍一下 Python 中函数式编程的几种形式

1. 高阶函数(Higher-order function):

编写高阶函数，就是让函数的参数能够接收别的函数，可从如下三种类型来更好的理解高阶函数。

» 函数本身也可以赋值给变量，变量可以指向函数。

```
>>> abs(-10)
10
>>> abs
<built-in function abs>
>>> f = abs
>>> f
<built-in function abs>
```

» 传入函数：变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

```
def add(x, y, f):
    return f(x) + f(y)
print(add(-5, 6, abs))
```

» 返回函数：高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

```
my_list = [1,2,3,6]
sum_func = lazy_sum(my_list)
print(sum_func())
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数，我们将求和函数赋值给 `sum_func`，只有当调用 `sum_func()` 时才真正触发了内部的计算（逻辑运行），这样的机制其实我们称为惰性计算

下面是一些 Python 中常用的高阶函数

```
# map 函数，对可迭代对象中的元素进行变化
```

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4]
squared_numbers = map(square, numbers)
print(list(squared_numbers)) # 输出: [1, 4, 9, 16]
```

```
# filter 函数，对可迭代对象中的元素进行过滤
```

```
def is_even(num):
    return num % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(is_even, numbers)
print(list(even_numbers)) # 输出: [2, 4, 6]
```

```
# sorted 函数，对可迭代对象内部元素按照某个元素进行排序
```

```
words = ['banana', 'apple', 'cherry']
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words) # 输出: ['apple', 'banana', 'cherry']
```

```
# reduce 函数，对可迭代对象中的元素两两规约(根据传入的函数)后放入，再继续合并
```

```
# 下面这个例子的过程可以认为是 1*2=2, 2*3=6, 6*4=24
```

```
from functools import reduce
```

```
def multiply(x, y):
    return x * y
```

```
numbers = [1, 2, 3, 4]
product = reduce(multiply, numbers)
print(product) # 输出: 24
```

```
# zip 函数，对两个可迭代对象进行组装，两两一对，要求两个对象的元素个数相等
```

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
```

```
paired = zip(names, ages)
print(list(paired)) # 输出: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

动手动脑

- » 上面我们已经展示过如何使用 reduce 对一个列表内的数字进行累乘，你能使用 reduce 对一个列表进行求和吗？
- » 使用 zip 可以同时对两个可迭代对象进行遍历，如下

```
my_list1 = [1,2,3]
my_tuple = [4,5,6]
for i,j in zip(my_list,my_tuple):
    print(i,j)
```

那么你能否自己实现一个简单的 zip 函数呢？

- » 相应的，我觉得你应该有能力手动实现 filter 和 map 函数了，下面给出接口

```
def my_filter(func,iterable_obj):
def my_map(func,iterable_obj);
```

2. 匿名函数：

- » 当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。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。
- » 用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数，同样，也可以把匿名函数作为返回值返回。

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

```
def build(x, y):
    return lambda: x * x + y * y
```

3. 装饰器：

- » 我们初始定义一个函数，希望增加它的功能但不希望修改其定义，在代码运行期间动态的增加功能的方式，称之为“装饰器”。本质上，decorator 就是一个返回函数的高阶函数。看下面的案例，我们希望在执行一个函数时候能打印该函数的名称

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

```
@log
def now():
    print('2024-6-1')
>>> now()
call now():
2024-6-1
```

下面我们来解释一下上面这段函数

我们首先定义了 `log` 函数，接受一个 `func` 的参数，

然后我们在 `log` 内部声明了一个 `wrapper` 函数，在里面调用了 `func`

然后我们声明了 `now` 这个函数，并使用了 `@log` 这个装饰器

当调用 `now()` 的时候，会先将 `now` 传入 `log` 中，然后按照 `log` 函数中的逻辑继续执行代码，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志。在这里你也可以认为调用了 `now` 就是调用了 `wrapper` 函数

动手动

» 编写一个修饰器，要求能够输出被修饰的函数的运行时间

关于函数式编程

函数式编程还有很多很多很多内容没有讲，这里也只是抛给大家一个引子，后面有机会可以自己去学习相关的内容哦。在这里大家只要将函数式编程简单理解为，你可以将函数作为一个变量赋值，当成一个参数去传递。

IV.4. 元编程

元编程就是关于创建操作源代码(比如修改、生成或包装原来的代码)的函数和类。主要技术是使用装饰器、类装饰器和元类。不过还有一些其他技术，包括签名对象、使用 `exec()` 执行代码以及对内部函数和类的反射技术等。简单说，你可以通过元编程技术来定制化你的源代码化行为。简单来说，元编程就是在程序运行时，动态的修改程序的内容

在 Python 中，元编程主要依赖于以下几个核心概念：

- » 动态类型：Python 是一种动态类型语言，这意味着变量的类型在运行时可以更改。这种灵活性使得 Python 代码可以在运行时动态地创建、修改和删除类和方法。最开始的时候我们曾提及过，对于一个变量你可以用任何类型的值对其进行复制，因此该变量的类型也会随其值而改变，这就是所谓的动态类型。
- » 装饰器 (Decorators)：装饰器是一种元编程工具，它允许开发者在不修改原始函数或类代码的情况下，为它们添加额外的功能。装饰器本质上是一个接受函数或类作为参数的可调用对象，并返回一个新的函数或类对象。(上文有提及)
- » 反射 (Reflection)：反射是指程序能够检查和修改其自身结构（如类、方法、变量等）的能力。Python 的内置函数和模块，如 `dir()`, `type()`, `getattr()`, `setattr()`, `callable()`, 以及 `inspect` 模块，都提供了反射功能。
- » 泛型：泛型是一种编程概念，通俗地说，它允许你编写可以处理多种数据类型的代码，而不需要为每种类型单独编写代码。

下面我们来重点介绍下反射和泛型

IV.4.1. 反射

» Python 的反射 (Reflection) 是指程序能够在运行时获取到自身的信息，并进行操作的能力。这包括获取对象的类、方法、属性等信息，以及动态地调用对象的方法和修改对象的属性等。

» 反射在 Python 中主要通过以下几种方式实现：

1. `type()` 和 `isinstance()` 函数：可以用来获取对象的类型或一个对象是否是某个类的实例。
2. `dir()` 函数：可以用来获取一个对象的所有属性和方法。
3. `getattr()` 函数：可以用来获取一个对象的方法或属性。如果指定的属性不存在，它会抛出 `AttributeError` 异常。
4. `setattr()` 函数：可以用来动态地给一个对象设置属性。如果属性不存在，它会创建这个属性。
5. `hasattr()` 函数：可以用来检查一个对象是否具有某个属性。
6. `delattr()` 函数：可以用来删除一个对象的属性。

» 反射使用的场景：

1. 动态导入模块
2. 动态获取对象属性
3. 动态调用对象的方法

看个例子吧

```
class MyClass:  
    def my_method(self):  
        return "Hello"  
  
obj = MyClass()  
method_name = 'my_method'  
if hasattr(obj, method_name):  
    method = getattr(obj, method_name)  
    print(method()) # 输出: Hello
```

我们来解释一下上面这段代码

首先声明了 `MyClass` 类，然后创建了 `obj` 对象。然后我们使用 `hasattr` 判断 `obj` 是否有 `method_name` 这个方法，如有，则使用 `getattr` 获取 `obj` 的方法，然后调用方法。假设我们不知道 `MyClass` 的内部实现，就需要使用这种形式去编写程序的逻辑。

IV.4.2. 泛型

Python 的泛型是类型提示(类型标注,类型声明)的一个高级特性，用于指示某些数据结构或函数可以接受多种数据类型，例如使得一个函数的参数能够接收多种类型的数据或者使得一个类能够接收并存储不同种类的数据。泛型在 Python 中通常通过模块 `typing` 实现。泛型的使用有助于提高代码的可读性和健壮性，尤其是在大型项目和团队协作中。

类型标注

前面我们提到了 Python 是动态型的语言，对任意变量的赋值是没有限制的。其实如果没有对其类型进行限定，其就是 Any 类型的变量。如果学过 c 语言，我们都知道在对变量(参数)声明时，必须需要声明其类型，就称为类型声明如下，我们声明了一个 int 型的变量 a

```
int a;
```

在 Python 中，没有强制我们必须声明类型，但我们同样可以使用 :类型 的方式对变量的类型进行声明，如下

```
a:int = 1
```

如果我们需要代码的可读性好一点是，通常是需要类型标注的，一旦进行了类型标注，就不可给该变量赋予其他类型的值。拿上面的这个例子来说，你已经声明 a 为 int，就不可为其赋予 str 类型的值

特别地，如果你需要对函数返回值的类型进行限定，可以使用 ->

```
def func() -> int:  
    pass
```

为什么需要泛型呢，事实上，Python 作为一个动态型的语言，不进行类型标注的话，完全不需要泛型。但是事实上，在构建项目时，是需要一定的类型标注的。事实上如果变量类型不可变，考虑以下问题

```
# 处理函数  
def process_item(item:int)->int  
    return item*item  
  
item = 3  
print(process_item(item))
```

上述代码定义了一个 process_item 的函数用于处理元素，接着我们传入了一个值为 3，类型为 int 的变量 item 进去，进行处理。如果此时我希望对一个类型为 float 的变量，也能进行相同的操作，你会如何处理呢，由于我们限定了 process_item 传入的参数只能是 int 型的，最直观的想法是再创建一个函数，功能和 process_item 相同，只不过传入的参数是 float 型的，如下

```
def process_item_float(item:float) -> float:  
    return item * item  
  
item = 3.0  
print(process_item_float(item))
```

但是这样是否太麻烦了呢？要创建两个函数确实麻烦。函数的逻辑都是相同，我能否只创建一个函数就解决所有类型问题呢？有的，就是使用泛型函数，在 python 中，如果要使用泛型就要使用 typing 模块，具体如下

```
from typing import TypeVar,  
  
T = TypeVar('T')
```

```
def process_item(item:T) -> T:  
    return T * T  
  
item1 = 2  
print(process_item(item1))  
item2 = 2.0  
print(process_item(item2))
```

我们来解释一下上面这个代码

首先我们使用 `from` 和 `import` 从 `typing` 中导入 `TypeVar` 函数，定义了一个泛型变量 `T`。接着我们使用 `T` 声明了一个泛型函数 `process_item`，`process_item` 可以接受任意类型的参数，对其进行逻辑处理，该例子中的逻辑是相乘。然后我们就可以传入任意类型的参数进去了，当然，如果传入的类型不支持 `*` 运算，是会报错的，比如你传入一个 `str` 型的值就会报错。

通过上面的方式，我们可以提高代码的复用率，当然，如果要使用泛型编程，你必须需要有足够的经验和严密的逻辑，不然传入一个不支持的类型，程序运行过程是会出错的。事实上，如果你有机会学习 `c++`，你会发现里面很多地方使用了泛型编程。

下面继续系统介绍两种泛型的使用

1. 创建泛型类，可以使用 `typing.Generic` 创建自定义的泛型类。

```
from typing import TypeVar, Generic  
  
T = TypeVar('T')  
  
class Stack(Generic[T]):  
    def __init__(self) -> None:  
        self.items: List[T] = []  
  
    def push(self, item: T) -> None:  
        self.items.append(item)  
  
    def pop(self) -> T:  
        return self.items.pop()  
  
stack = Stack[int]()
stack.push(1)
stack.push(2)
print(stack.pop()) #
```

在这个例子中，`Stack` 类是一个泛型类，可以用于存储任何类型的元素。在实例化时，你可以指定具体的类型，比如 `Stack[int]`。

2. 泛型函数，泛型也可以用于定义可以接受多种类型参数的函数。

```
from typing import TypeVar, Generic  
  
T = TypeVar('T')
```

```
def first(items: List[T]) -> T:  
    return items[0]  
  
print(first([1, 2, 3])) # 1  
print(first(["apple", "banana", "cherry"])) # apple
```

在这里，`first` 函数可以接受任何类型的列表，并返回列表中的第一个元素。

注意事项

1. 类型检查：Python 的类型提示仅用于静态类型检查，不会影响运行时行为。
2. 兼容性：确保使用泛型的代码与使用的 Python 版本兼容。
3. 可读性：合理使用泛型以提高代码的可读性和维护性，避免过度复杂化。

V. 工具链

V.1. 调试器

`pdb` 是 Python 的内置调试器(不需要额外下载)，提供了一种互动式调试 Python 程序的方式。它允许开发者逐步执行代码、检查变量、设置断点等，从而帮助定位和修复程序中的错误。以下有关 `pdb` 的一些关键点：

1. 逐步执行：可以逐行执行代码，帮助理解程序的执行流程。
2. 设置断点：可以在代码的特定行设置断点，当程序执行到该行时会暂停，方便检查状态。
3. 检查变量：可以动态查看和修改变量的值，帮助调试逻辑问题。
4. 调用堆栈检查：可以查看当前的调用堆栈，帮助理解函数调用的上下文。

建议

有时候，难免会出现一些涩晦的名词，这是一些后续大家才会接触的内容。我们在能力范围内尽可能地让大家能够入门并学会思考，但当你觉得某些东西看不懂，在不影响你后续学习的情况下，大可先跳过去，不需要强行记忆。

V.1.1. PDB 常用命令

- » `n` (next)：执行下一行代码。
- » `c` (continue)：继续执行，直到下一个断点。
- » `s` (step)：进入当前行的函数调用。
- » `q` (quit)：退出调试器。
- » `p` (print)：打印变量的值，例如 `p my_variable`。
- » `l` (list)：列出当前执行位置附近的代码行。
- » `b` (break)：设置断点，例如 `b 12` 在第 12 行设置断点。

使用示例：

要在 Python 程序中使用 `pdb`，可以按以下方式引入：

```
import pdb

def example_function(x):
    pdb.set_trace() # 设置断点
    y = x + 1
    return y

result = example_function(5)
print(result)
```

1. 启动调试器

运行程序时，执行到 `pdb.set_trace()` 时会进入调试模式。你可以使用上述命令来调试程序。

2. 命令行调试

你也可以直接在命令行中运行 Python 程序并启动 `pdb`：

```
python -m pdb myscript.py
```

这会在程序执行时启动调试器。

尝试调试你的程序

- » 任意选择你前面写给过的程序，试着用 `pdb` 设置断点，查看信息
- » 目前很多 IDE(Integrated Develop Environment 集成开发环境)都自带调试的功能（如 Pycharm），VS code 作为一款编辑器也具备调试的功能，具体的做法可以参考[VS code 中使用 python](#) 中的调试代码部分
- » 除了 `pdb` 之外，还有很多更优秀调试的工具，请 STFW，选择一款适合你的工具吧

V.2. 包管理

包管理是指组织、安装、更新和删除软件包（即程序和库）的一种方法和系统。通俗来讲，如果你曾使用过类似软件管家的软件，可以简单认为包管理工具的功能类似于软件管家。我们可以通过软件管家下载、删除、更新软件。类似地，可以通过包管理工具下载、删除、更新软件包。

`pip` 是 Python 的一个包管理工具，用于安装和管理 Python 程序包（也称为库或模块）。它使得开发者能够轻松地下载、安装和更新第三方库，这些库可以帮助你在项目中实现各种功能，而不需要从头编写所有代码。在我们安装好 Python 时，`pip` 也已经随之安装好了。

1. 安装工具：想象一下，`pip` 就像是一个超市的购物车。你可以把想要的商品（即 Python 库）放入购物车，并通过 `pip` 把它们带回家（安装到你的计算机上）。
2. 库的来源：`pip` 从 Python 包索引（PyPI）下载库。PyPI 就像是一个巨大的在线商店，里面有数以千计的 Python 库供你选择。

尝试一下

打开终端，输入 `pip --help` 来查看 `pip` 的功能和使用方法，我们可以看到终端中输出了大量的信息，告诉我们 `pip` 命令的可选参数，以及每个参数的含义。

建议

- » 要习惯在终端中这样的交互方式，之后当我们使用 Linux 操作系统的时候，很多命令都是通过命令行执行的
- » 类似于 pip --help 这样“命令”+“参数”的命令有很多，之前我们执行文件使用的指令 python main.py 是“命令”+“目标文件”的形式，之后我们还会遇到“命令”+“参数”+“目标文件”这样冗长的指令

V.2.1. PIP 简单操作

1. 安装库: 使用 pip 安装库非常简单。比如，想要安装一个名为 requests 的库，只需要在终端中输入：

```
pip install requests
```

2. 更新和卸载: 除了安装，pip 也可以用来更新库和卸载不再需要的库。例如：

» 更新库： pip install --upgrade requests

» 卸载库： pip uninstall requests

3. 管理依赖: 在一个项目中，可能会用到多个库。pip 允许你将这些依赖项列在一个文件中（通常是 requirements.txt），这样其他人可以轻松地安装你项目所需的所有库：

```
pip install -r requirements.txt
```

安装一些常用的库

使用 pip 安装 numpy,matplotlib,scikit-learn 等常见的第三方库

安装不了?!

当你使用 pip 的时候可能会卡在 search 的这一步，这是由于 PyPI 的资源网站位于国外，访问时间非常长，我们可通过配置 pip，使得 pip 从国内的镜像源网站下载第三方库，具体请看下一小节

V.2.2. 更换库源

前面我们提到，pip 通过在 PyPI 中搜索满足要求的第三方库(模块)，但是这样产生的体验并不好，因为 PyPI 资源网站位于国外，访问时间非常长，我们可以配置 pip 的源库为国内的镜像网站，具体可以参考 [pip 配置清华源](#)

配置镜像源

- » 我们可以通过 pip install 库名称 -i 镜像网站 的方式来临时指定镜像源下载某一个库，但这样极不方便
- » 尝试将 pip 的库源永久设定为清华源

V.3. 虚拟环境管理

虚拟环境是一种隔离的工作空间，可以让你在不同项目中使用不同的依赖和 Python 版本，而不会相互干扰。为什么需要虚拟环境？试着想一下这样的场景，当你的 requests 库需要 numpy 的版本 ≥ 1.1 而 matplotlib 库要求 numpy 的版本小于 < 1.1 ，这就是版本冲突。

就好比，在同一个厨房里，如果我们限定一个厨房只能有一把刀(numpy)，一个厨师要求只使用 13cm 长及以上的刀(requests)，另一个厨师要求只能使用 13cm 以下的刀(matplotlib)，这样的情况两个厨师长不得不打一架(版本冲突)？

那么我们要如何解决？最好的方式自然是将两个厨师放在不同的厨房里。对应地，就是使用虚拟环境，将 requests 所需要的依赖和 matplotlib 需要的依赖隔离到两个不同的工作空间，在不同的虚拟环境当中安装对应的依赖库（因此，你可以认为虚拟环境就是一个个独立工作的厨房）。下面我们分别介绍 python 的两种虚拟环境的方案：venv 和 conda

V.3.1. VENV

venv 是 Python 3 中自带的用于创建虚拟环境的模块，不需要额外安装，下面我们简单介绍一下如何使用 venv 的虚拟环境。

1. 创建虚拟环境：在项目目录中，使用以下命令创建虚拟环境，其中 myenv 是你要创建的虚拟环境的名字

```
python -m venv myenv
```

2. 激活虚拟环境：创建虚拟环境之后，使用 activate 文件激活虚拟环境，不同操作系统的方式不太一样，激活后，你会在命令提示符中看到虚拟环境的名字，表明你已进入该环境。

» Windows:

```
myenv\Scripts\activate
```

» macOS/Linux:

```
source myenv/bin/activate
```

3. 安装依赖：在激活的虚拟环境中，你同样可以使用 pip 安装所需的库。例如：

```
pip install requests
```

4. 退出虚拟环境：可以通过 deactivate 命令退出虚拟环境：

```
deactivate
```

体验 venv

试着使用 venv 创建一个虚拟环境，并在该环境下运行你的程序

多个虚拟环境

在你的项目下继续创建一个虚拟环境，并尝试在多个虚拟环境之间来回切换

建议

手册不一定总是对的，因为编写手册的过程难免会出现笔误的情况，或者由于本身对这块知识点比较模糊而给出错误的教程或解释，当你反复尝试书册上面的案例而无法成功时，请自行 STFW，并向我们提出反馈，感谢。

V.3.2. CONDA

Conda 是一个开源的包管理和环境管理系统，旨在简化软件包的安装和管理，特别是针对数据科学、机器学习和科学计算领域的用户。以下是关于 Conda 的一些关键点：

1. 包管理:Conda 允许用户轻松安装、更新和卸载软件包。它支持多个语言的包，包括 Python、R、Ruby、Lua、Scala、Java、JavaScript、C/C++、FORTRAN 等。
2. 环境管理:Conda 允许用户创建和管理独立的环境，每个环境可以有不同的依赖包版本。这对于避免依赖冲突非常重要。
3. 跨平台:Conda 支持 Windows、macOS 和 Linux 等多个操作系统。

Miniconda 是一个 Conda 的轻量级的发行版，只包含了必要的工具和基础包，请到 [Miniconda 下载官网](#) 中下载 Miniconda，具体的安装过程请参考 [Miniconda 安装教程](#)

安装 Miniconda

请跟随教程安装 Miniconda

conda 与 venv

相比于 venv，conda 是比较臃肿的，但是对于新手来说操作比较简单，当然还有很多虚拟环境的方案，请 STFW 了解更多，并选择一种适合自己的虚拟环境方案吧

V.3.2.1. CONDA 常用操作

1. 创建新环境:其中 myenv 为虚拟环境的名称，同时可以指定虚拟环境中 Python 的版本

`conda create --name myenv python=3.9`

2. 激活环境:激活后，你会在命令提示符中看到虚拟环境的名字，表明你已进入该环境。

`conda activate myenv`

3. 退出虚拟环境:

`conda deactivate`

4. 安装包:

`conda install numpy`

5. 更新包:

`conda update numpy`

6. 列出所有环境:

`conda env list`

7. 删除环境:其中 myenv 为你要删除的环境的名称

`conda remove --name myenv --all`

使用 conda 创建虚拟环境

- » 类似于 pip，你同样可以使用 `conda --help` 来查看关于 conda 命令的使用方法和各种参数的含义。事实上，绝大多数命令都可以使用命令 `--help` 的方式来查看对应的帮助文档，之后当你使用 Linux 系统时，你不可能一下子记住所有命令的使用方法，需要经常使用 `--help` 参数。
- » 现在打开终端，使用 conda 创建一个虚拟环境吧。

安装 conda 之后

安装 conda 并配置好环境变量之后，请重新打开终端。你可以看到命令行提示符最前面出现 (base) , 这就是你当前所处的虚拟环境， base 可以认为是你最开始安装 Python 的环境。需要注意的是，如果要切换虚拟环境，我们建议先使用 conda deactivate 退出当前虚拟环境，再通过 conda activate 激活新的虚拟环境，否则可能会环境嵌套的问题，就比如，你可能看到，命令行提示符显示 (base)(myenv) 这样的多种环境堆叠的情况

conda 与 pip

可以看到 conda 也可以承担安装第三方库的角色，因此 conda 也是包管理工具的一种，记住，包管理工具会记录所有已安装的第三方库。但是这样会产生一个问题，同一个虚拟环境，存在两个包管理工具，如果我们需要安装 numpy 这个第三方库时，是选择 conda install numpy 还是 pip install numpy 呢？这里同样会出现 版本冲突 的问题

其实在安装第三方库时，包管理工具会根据当前环境的依赖要求，选择合适版本的库来安装。比如，当前环境下的 conda 查看已安装的第三方库的依赖要求(通过 conda install 安装的)，要求 numpy ≥ 1.1 ，那么 conda 就会选择符合要求版本的 numpy 安装，但是如果同时环境通过 pip 安装的第三方库要求 numpy < 1.1 ，那么最终，无论安装那个版本的 numpy，都无法同时满足两个包管理的要求。

根据经验，我们推荐的做法是，如果能够先使用 conda 安装第三方库，先使用 conda，如果 conda 无法寻找到合适的库，再使用 pip。当然另外一种做法是，只把 conda 当成虚拟环境工具，使用 conda 创建虚拟环境之后，只使用 pip 安装第三方库。以上两种做法遇到的问题是比较少的，千万不要交换地使用两种工具安装!!!

到这里，我们就完成了 python 的入门，在此期间我们也渗透性的教大家一些编程范式。当然大家思考问题的方式和解决问题的能力，才是最宝贵的财富。如果你对本学习手册有任何建议，请反馈给我们，期待我们在下一个部分见面！