霍格沃兹测试开发学社-Python基础教程

Python基础教程

霍格沃兹测试开发学社

None

Table of contents

1. Python编程语言

1.1 L1.Python语法与数据结构

1.1.1 Python 介绍与环境配置

初识Python

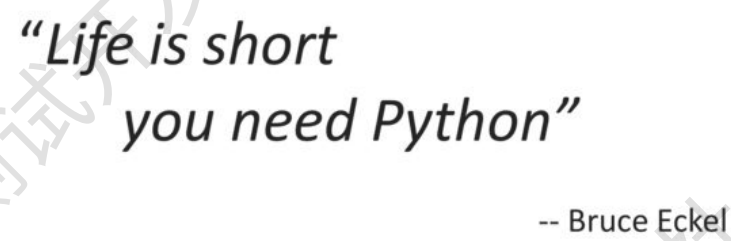
初识 PYTHON

Python 发展历史

Python 是一门高级编程语言，由 Guido van Rossum（龟叔） 在 1989 年发明，设计 Python 语言的初衷是为了创造一种介于 C 和 shell 之间，简洁方便，易学易用，功能全面，可拓展的语言。

龟叔在 1989 年圣诞节期间，以由荷兰的数学和计算机研究所开发的 ABC 语言为蓝本，开始开发一门新的编程语言，目标让新语言即能像 C 语言一样能够全面调用计算机的功能接口，又可以像 shell 一样可以轻松的编程，并且以龟叔所挚爱的电视剧 `Monty Python's Flying Circus` 命名新语言为 Python。

龟叔作为一个语言设计爱好者，曾经有过设计语言的尝试。这一次，也不过是一次纯粹的 hacking 行为。



Python一经问世，其简洁方便的编程方式，受到编程者的欢迎和喜爱，并吸引了大量的开发者使用，而且加入到 Python 的开发维护中。

龟叔维护了一个mail list，Python用户通过邮件进行交流，对 Python 进行拓展或改造。随后这些用户将改动发给龟叔，并由龟叔决定是否将新的特征加入到Python或者标准库中。

如果代码能被纳入Python自身或者标准库，这是极大的荣誉。由于龟叔至高无上的决定权，他因此被称为“终身的仁慈独裁者”。

Python 自发布以来，主要经历了三个版本的变化

- Python 1.0 版本 1994 年发布(已过时)
- Python 2.0 版本 2000 年发布 (到 2018 年 3 月已经更新到 2.7.14，目前停止维护)
- Python 3.0 版本 2008 年发布 (目前已经更新到 3.12)。

从 2004 年开始，Python 的使用率呈线性增长，2021 年，Python 再次荣膺 TOBIE 榜单首位！这一成就再次彰显了 Python 的强大影响力和广泛应用，也印证了 Python 作为全球最流行的编程语言的地位。

Oct 2021	Oct 2020	Change	Programming Language	Ratings	Change
1	3	↑	Python	11.27%	-0.00%
2	1	↓	C	11.16%	-5.79%
3	2	↓	Java	10.46%	-2.11%
4	4		C++	7.50%	+0.57%
5	5		C#	5.26%	+1.10%
6	6		Visual Basic	5.24%	+1.27%
7	7		JavaScript	2.19%	+0.05%
8	10	↑	SQL	2.17%	+0.61%

Python 特点

Python是一种简单、易读、易学和高效的编程语言，具有以下特点：

1. 简单易学：Python采用清晰简洁的语法，注重代码的可读性和可维护性，使得初学者能够快速上手并编写出清晰的代码。
2. 面向对象：Python是一种面向对象的编程语言，支持封装、继承和多态等面向对象的概念，可以更好地组织和管理代码。
3. 开放源代码：Python是开源的，拥有庞大的开发者社区，因此可以方便地获得开源库和模块，可以加速开发过程并减少重复劳动。
4. 跨平台：Python可以在多个操作系统上运行，包括Windows、Mac OS和Linux等，具有很强的跨平台性。
5. 大量的库和框架：Python拥有丰富的标准库，覆盖了各种常用的功能模块。此外，Python还有大量的第三方库和框架，如NumPy、Pandas、Django等，可以满足各种不同领域的需求。
6. 强大的数据处理能力：Python提供了很多用于数据处理和科学计算的库，如NumPy、Pandas和Matplotlib等，使得数据分析和数据科学变得更加简单和高效。
7. 可扩展性：Python可以与其他语言进行无缝集成，可以轻松地扩展功能，使用C语言编写的扩展模块可以通过Python的接口调用。

应用领域

Python 作为一种功能强大且简单易学的编程语言而广受好评，并且在不同的应用域大放异彩，比如：

- Web 开发
- 大数据处理
- 数据分析
- 人工智能
- 自动化运维开发
- 云计算
- 爬虫
- 游戏开发
- 自动化测试
- 测试开发

目前，由于大数据、人工智能(ChatGPT 等 AI 技术)的流行， Python 变得比以往更加流行 。

如何学习

1. 由浅入深：如果是有基础的同学，可以挑自己感兴趣的内容学习。如果没有基础的同学，跟着大纲一步步学习，切忌跳着学习。

2. 利用好碎片时间：在琐碎时间结合视频配套的教程进行学习，学完一个章节务必要完成综合练习与实战练习，以检验学习的效果。

3. 实战练习：学完一个章节务必要完成综合练习与实战练习，以检验学习的效果。

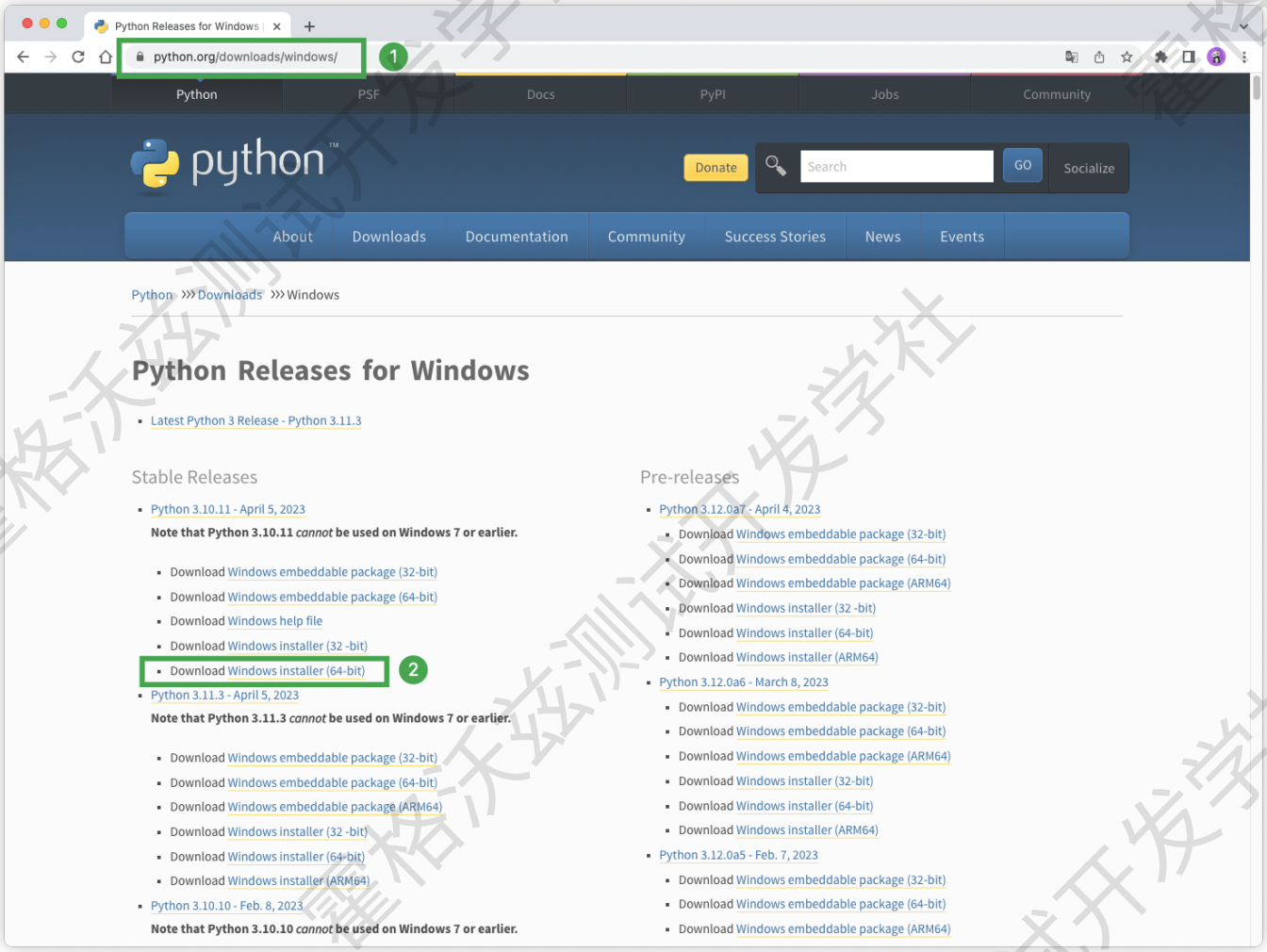
安装Python（Windows）

安装 PYTHON（WINDOWS）

下载 Python 解释器

下载地址

通过下载页面，可以在该页面上看到下载链接。



在下载列表中以“(64-bit)”结尾的链接是 64 位的 Python 安装程序，以“(32-bit)”开头的链接是 32 位的 Python 安装程序 。现在大部分电脑的 Windows 系统都是 64 位的。

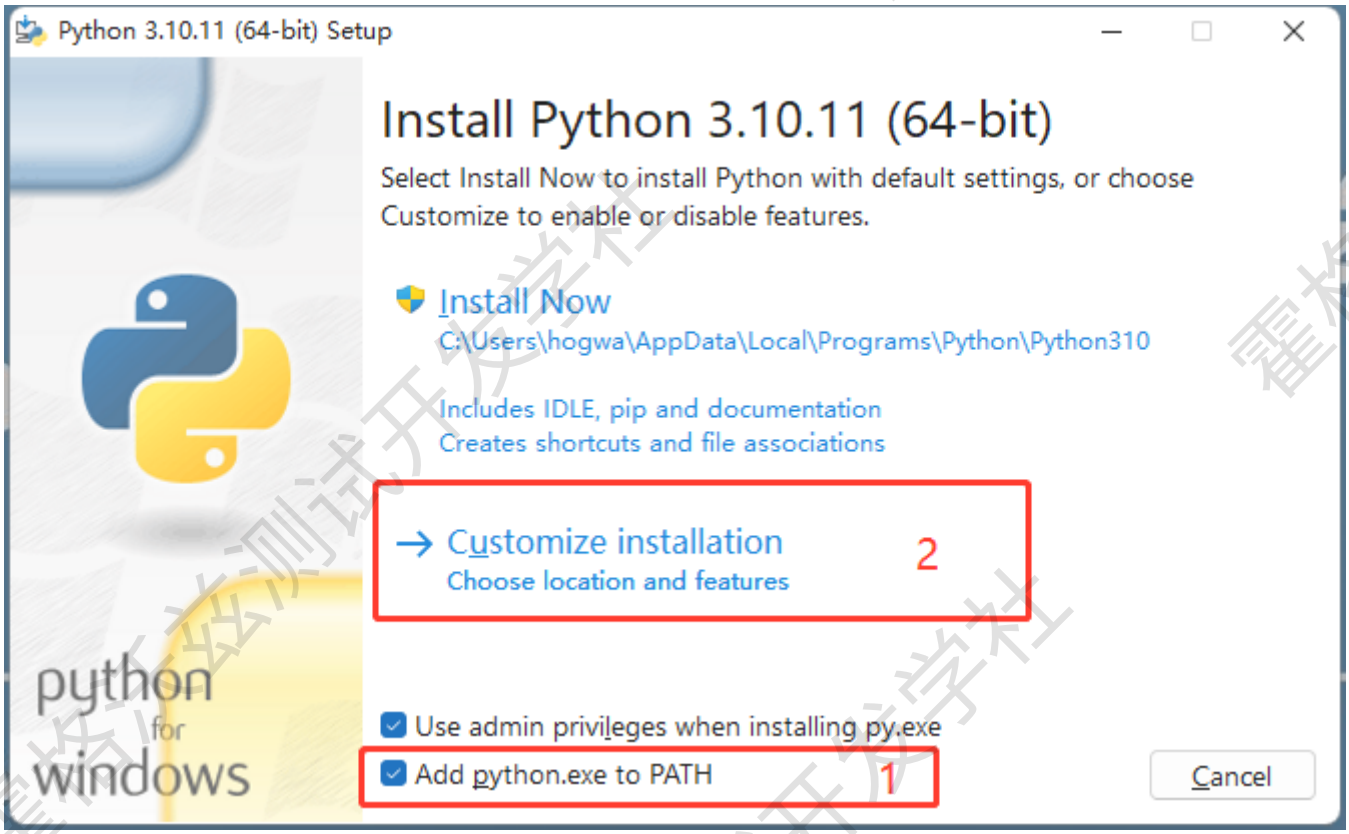
需要注意的是，此安装包已经不再支持 Windows7 系统版本，最好是使用 Win10 或者更新的版本。

下载完成后会得到 `Python-3.10.11-amd64.exe` 安装文件 。

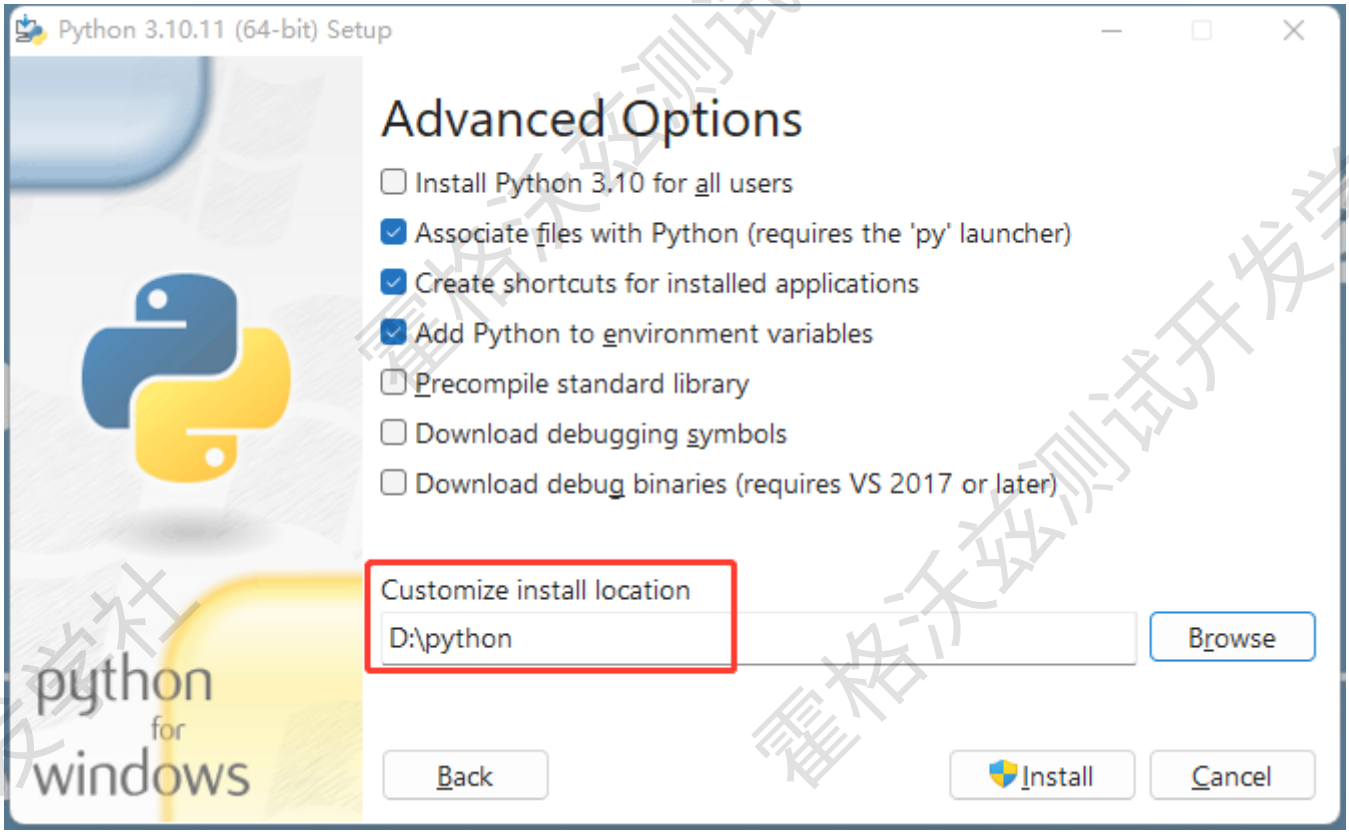
安装 Python 解释器

1. 双击 `Python-x.x.x-amd64.exe` 文件，系统将会开启 Python 安装向导
2. 勾选 `Add Python.exe to PATH` 复选框 ，可以将 Python 命令工具所在 目录 添加到系统 `Path` 环境变量中

3. 单击 `Customize installation`，可以在安装时指定自定义的安装路径。 单击该选项即可开始安装。



4. 在接下来的安装界面中只要改变 Python 的安装路径(建议安装在指定盘符的根路径下，比如 安装在 `D:\Python` 目录下)， 其他地方使用默认选项即可。



检查安装结果

安装完成后，启动 Windows 的命令行 `cmd` 程序 ， 在命令行窗口中输入 `Python` 命令(字母 p 是小写的)。

如果出现 Python 提示符(>>>)，就说明安装成功了 。

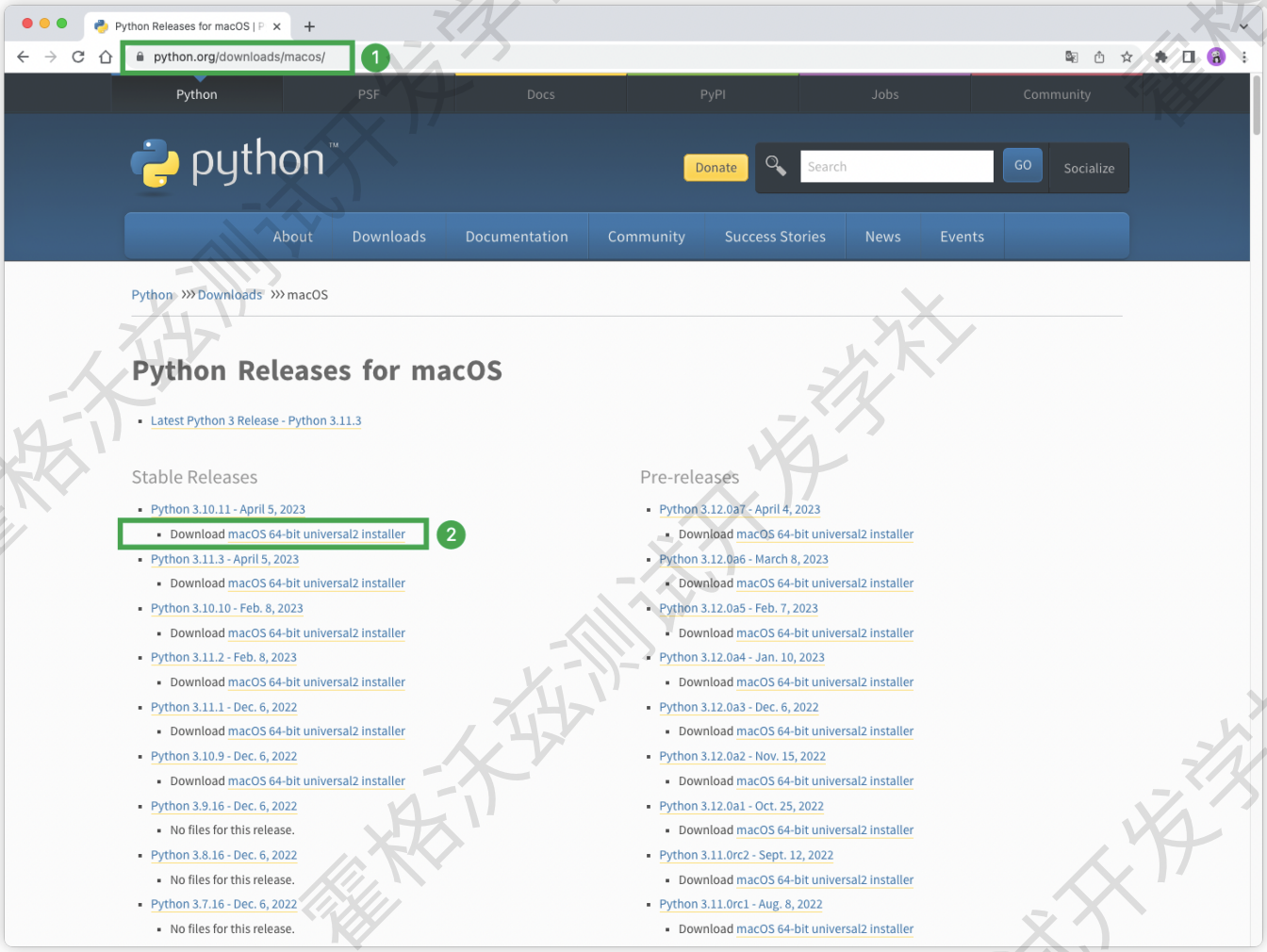
安装Python（macOS）

安装 PYTHON（MACOS）

下载 Python 解释器

下载地址

通过下载页面，可以在该页面上看到下载链接。



下载完成后会得到 `Python-3.10.11-macos11.pkg` 安装文件 。

安装 Python 解释器

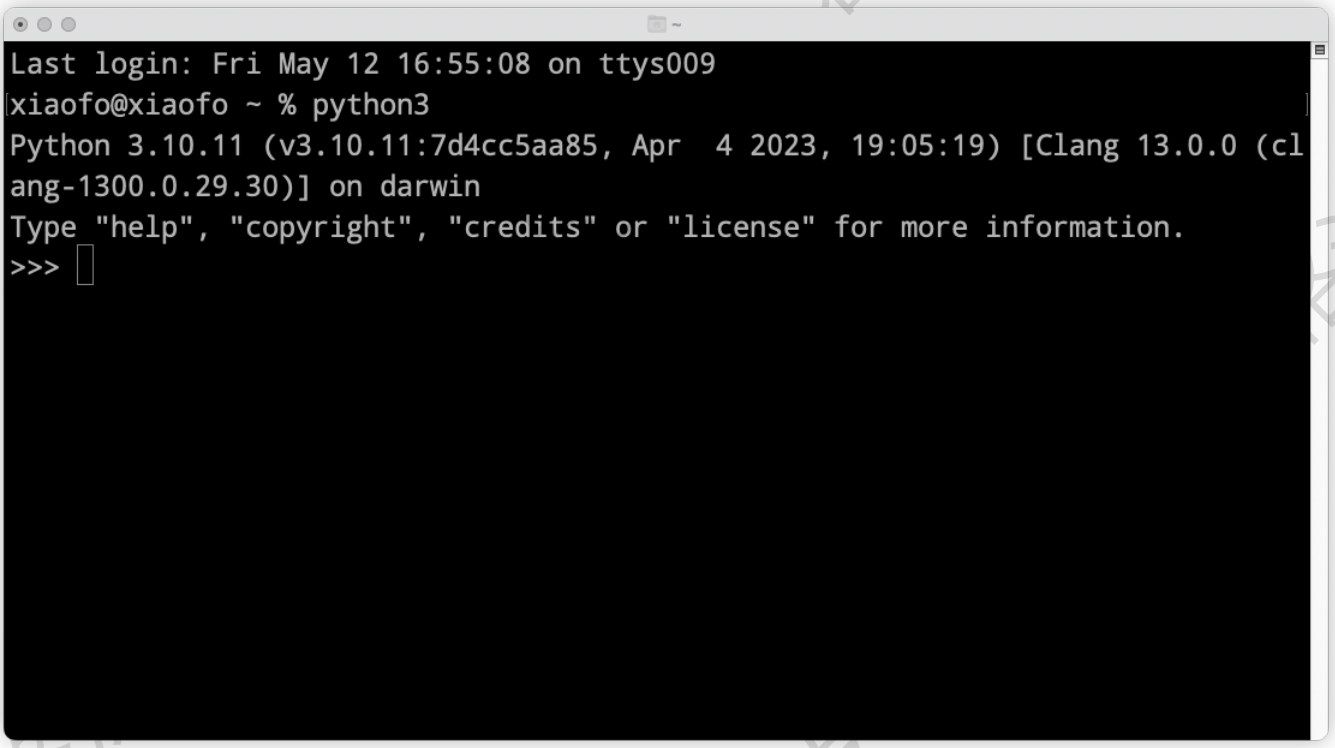
1. 双击 `Python-3.10.11-macos11.pkg` 文件，系统将会开启 Python 安装向导，如图所示

根据系统提示，点击“继续”或“确认”即可。

检查安装结果

安装完成后， 启动 `terminal` 终端 ， 在命令行窗口中输入“Python3”命令(字母 p 是小写的)。

如果出现 Python 提示符(>>>)，就说明安装成功了 。


~
Last login: Fri May 12 16:55:08 on ttys009
xiaofo@xiaofo ~ % python3
Python 3.10.11 (v3.10.11:7d4cc5aa85, Apr 4 2023, 19:05:19) [Clang 13.0.0 (cl
ang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

安装Pycharm

安装PYCHARM

PyCharm 简介

工欲善其事，必先利其器。为了良好的学习体验，我们需要一款功能全面，并且容易上手的代码编辑器，那么首选大名鼎鼎的 PyCharm。

PyCharm 是一款功能强大的 Python 集成化开发工具，具有跨平台性。

下载 PyCharm

打开 PyCharm 官网，这里提供了两个版本。第一个版本是 Professional（专业版本），这个版本功能更加强大，主要是为 Python 和 web 开发者而准备，是需要付费的。

第二个版本是社区版（Community），它是一个专业版的精简版，比较轻量级，主要是为 Python 初学者准备的。

由于专业版（Professional）需要激活，并且社区版（Community）已经包含了我们所需要的基本功能，所以这里我们选择社区版（Community）下载。

下载地址

另外，在该页面选择你电脑所对应的系统（Windows、macOS 、Linux）。

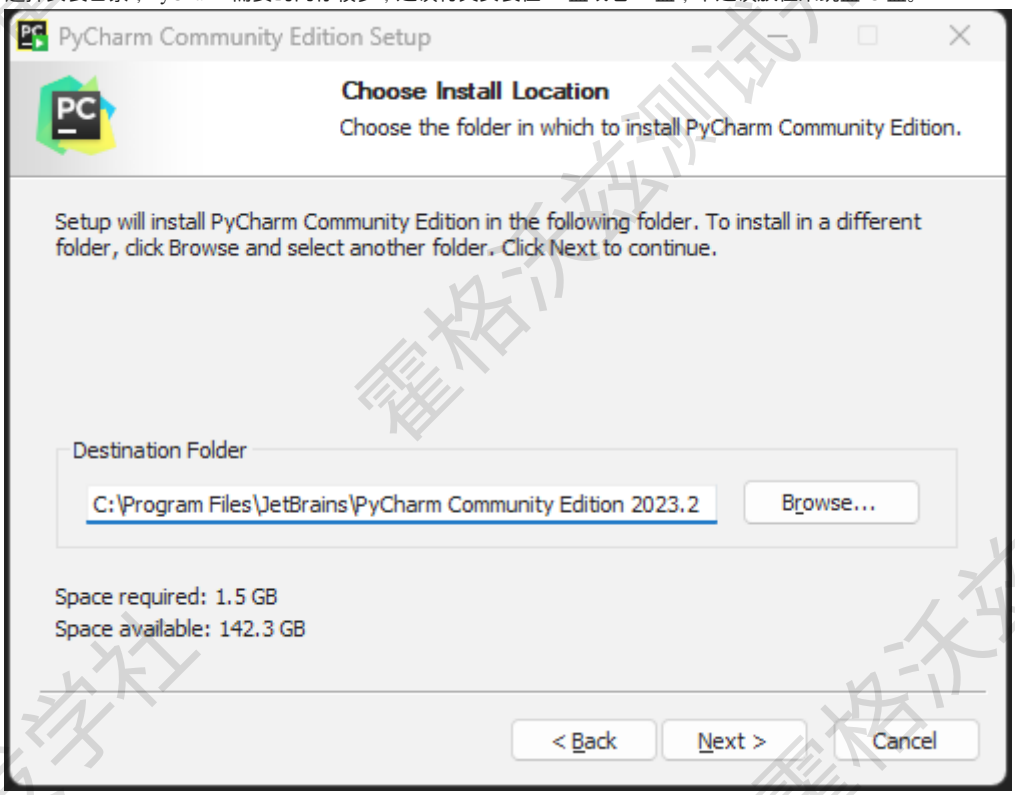
安装 PyCharm

Windows 系统安装

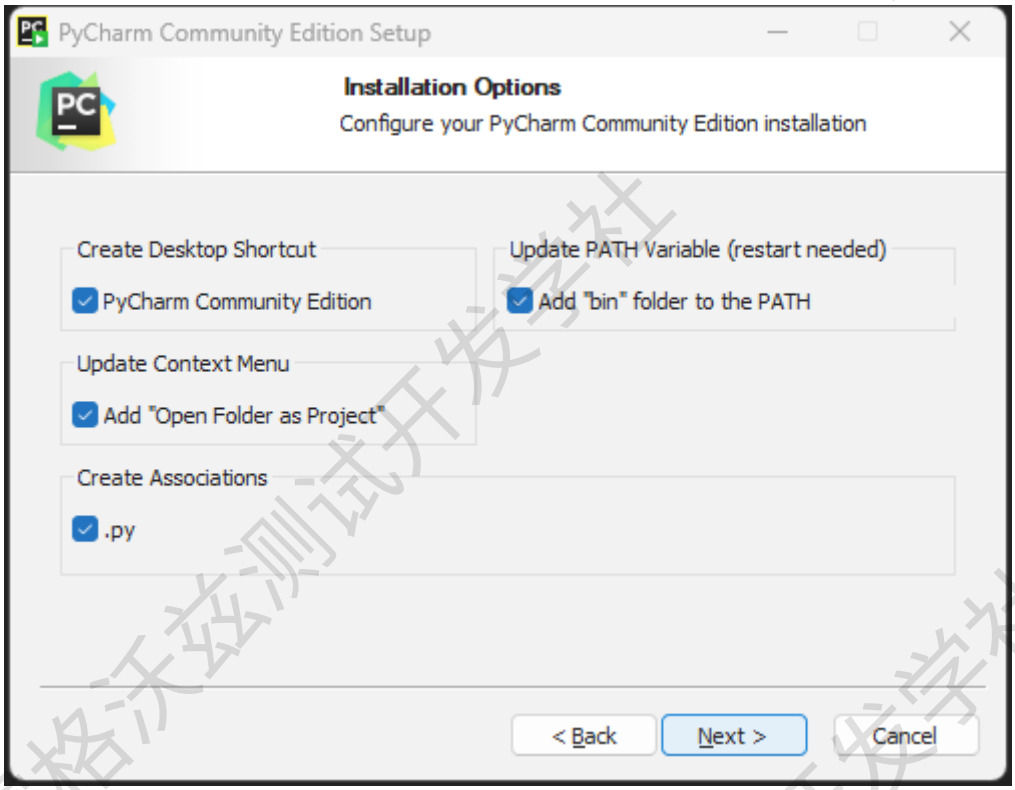
这里首先以 Windows 系统为例，安装 PyCharm 社区版（Community）。

1. 双击已下载的 PyCharm 安装包，出现如下图所示的界面，点击“next”。

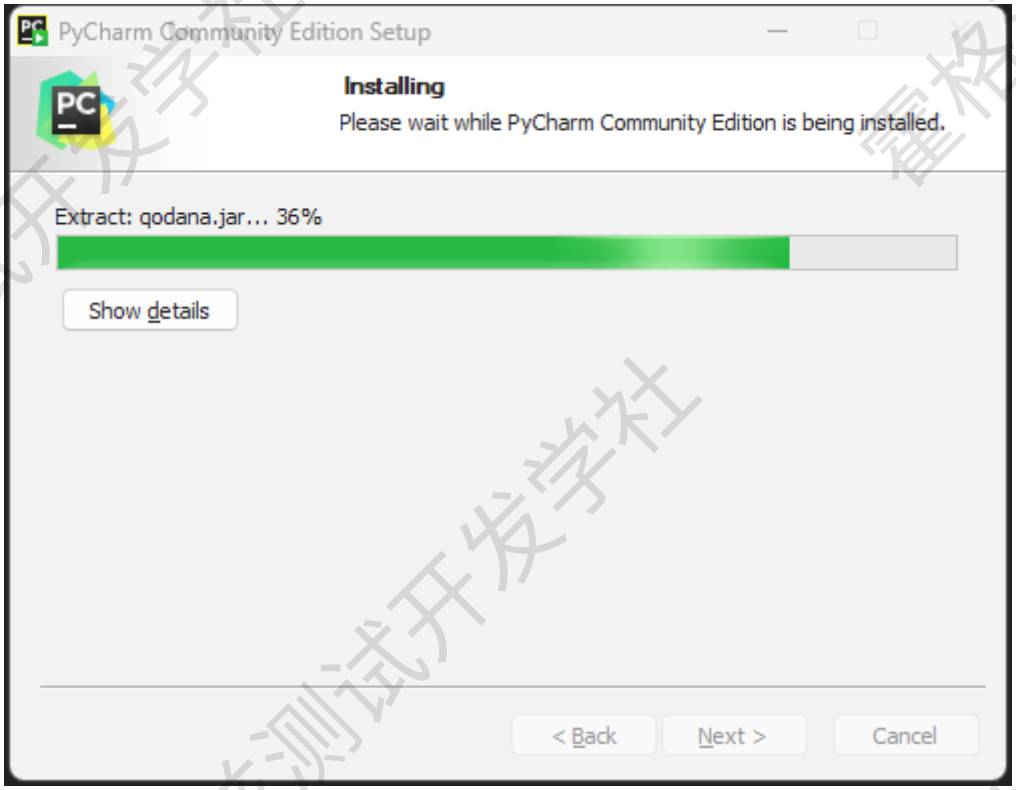
2. 选择安装目录，PyCharm 需要的内存较多，建议将其安装在 D 盘或者 E 盘，不建议放在系统盘 C 盘。

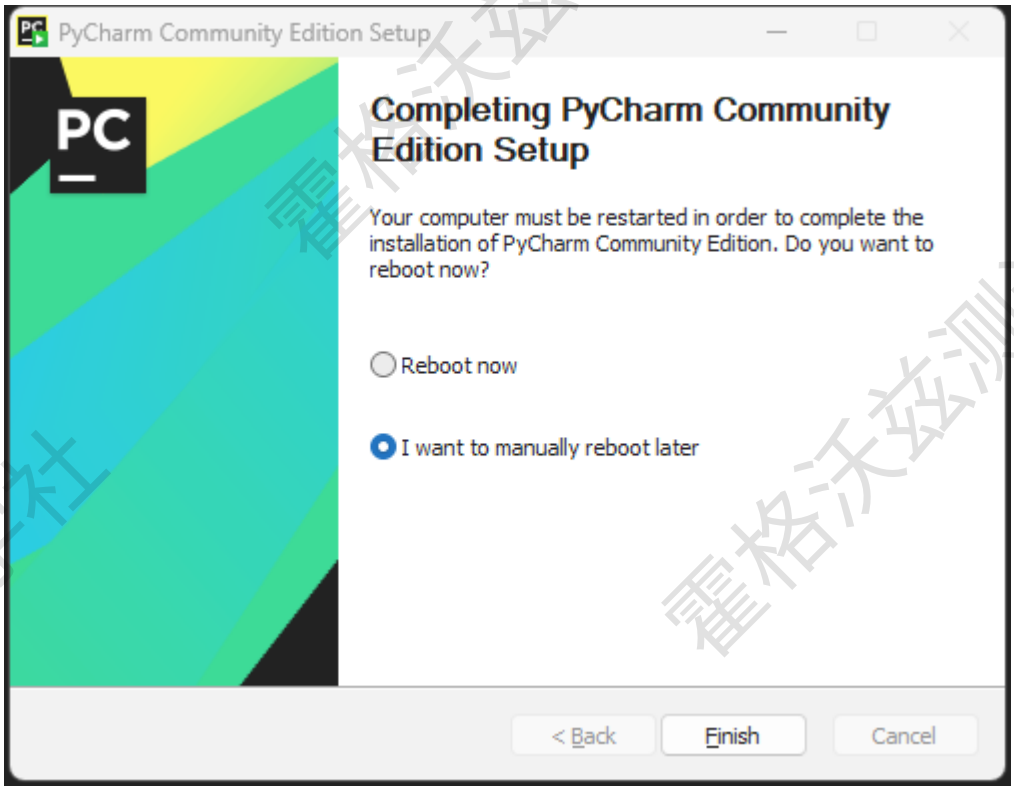


3. 进行相关设置，如果你无特殊需要按照图中勾选即可。



其中：

1. `create desktop shortcut`（创建桌面快捷方式），系统 32 位就选 32-bit,系统 64 位就选 64-bit。当前演示的电脑是 64 位系统，它自动显示 64 位。
2. `update path variable(restart needed)` 更新路径变量(需要重新启动)，`Add "bin" folder to the PATH`（将bin目录添加到路径中）。
3. `Update Context Menu`（更新上下文菜单），`Add "Open Folder as Project`（添加打开文件夹作为项目）。添加鼠标右键菜单，使用打开项目的方式打开此文件架。如果你经常需要下载一些别人的代码查看，可以勾选此选项，这会增加鼠标右键菜单的选项。也就是你双击你电脑上的 py 文件，会默认使用 PyCharm 打开。
4. `Create Associations` 创建关联，关联 `.py` 文件。将所有 `py` 文件关联到 PyCharm。
5. 

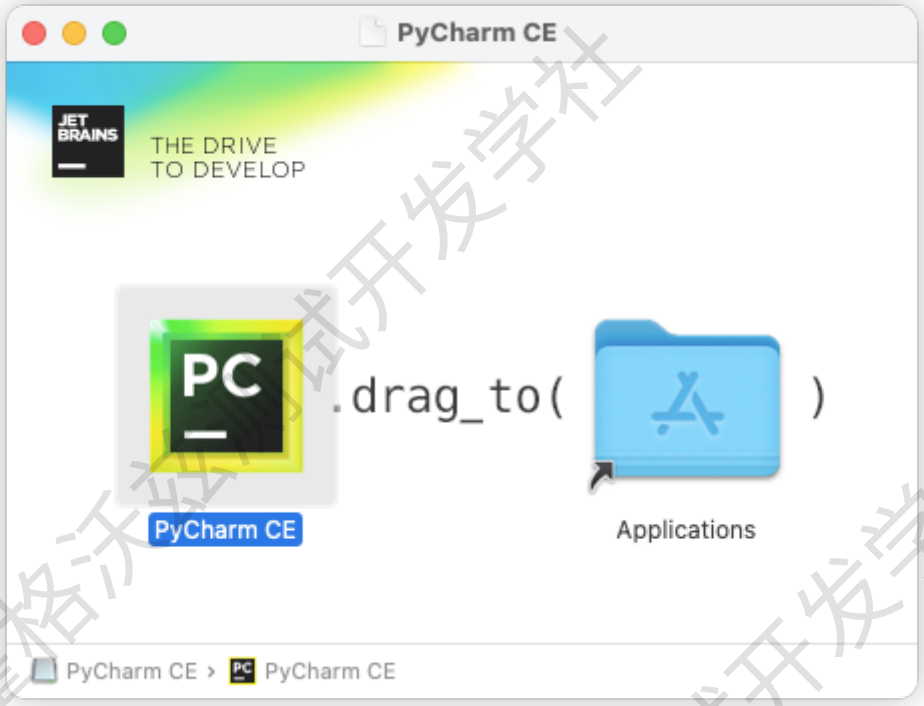
默认即可，点击 install。然后等待片刻。
6. 

安装完成

安装完成后，提示是否立即重启电脑，可以选择“稍后重启”，点击“Finish”即可。

macOS 系统安装

1. 双击运行下载好的 dmg 安装包，例如 `PyCharm-community-2023.1.1-aarch64.dmg`。 这时候后打开安装界面，如下图：



按住鼠标左键，将左侧 PyCharmCE 的图标拖拽到右侧的 Applications 图标上，等待片刻，即可自动完成安装。

第一个Python程序

第一个 PYTHON 程序

通过程序输出 `Hello World` 是在学习每一门编程语言时，都会接触到的第一个程序。

在 Python 中，可以通过内置函数 `print()` 实现向控制台输出 `Hello World` 。

使用 print()输出

可以进入 `命令行交互模式` 或使用 `PyCharm` 编写代码输出。

- 命令行交交互模式 实时输出程序执行结果，适合简单的逻辑或运算。
- PyCharm开发工具 方便组织管理代码，有代码提示功能，适合复杂逻辑或大型程序的开发。

实战练习

分别使用 `命令行交互模式` 和 PyCharm 编写程序打印： `测试开发进阶，首选霍格沃兹！`

Pycharm常用快捷键

PYCHARM常用快捷键

PyCharm 中提供了很多快捷键，利用这些快捷键，可以大大提升开发效率。

常用快捷键

说明：此文档以 Windows 系统为标准，MacOS 系统大部分情况只需将 `Ctrl` 替换为 `Command`。

快捷键	作用
Tab键	跳制表域
Ctrl + /	代码注释
Ctrl + F	查找
Ctrl + R	替换
Alt + Enter	问题修复
Ctrl + Alt + L	代码格式化
Ctrl + D	复制当前光标所在行代码
Ctrl(Alt) + Shift + 上下方向键	上下移动光标所在行代码
Ctrl + G / Command + L	跳转到指定行：列

官方手册

更多快捷键可以查看 PyCharm 帮助手册或查看下图。

- 帮助手册查找路径 `菜单-> Help -> Keyboard Shortcuts PDF`
- 快捷键图表 **Windows版**：

PyCharm

Find any action inside the IDE	Ctrl + Shift + A

CREATE AND EDIT

Show intention actions	Alt + Enter
Basic code completion	Ctrl + Space
Smart code completion	Ctrl + Shift + Space
Type name completion	Ctrl + Alt + Space
Complete statement	Ctrl + Shift + Enter
Parameter information / context info	Ctrl + P / Alt + Q
Quick definition	Ctrl + Shift + I
Quick / external documentation	Ctrl + Q / Shift + F1
Generate code	Alt + Insert
Override / implement members	Ctrl + O / Ctrl + I
Surround with...	Ctrl + Alt + T
Comment with line comment	Ctrl + /
Extend / shrink selection	Ctrl + W / Ctrl + Shift + W
Optimize imports	Ctrl + Alt + O
Auto-indent lines	Ctrl + Alt + I
Cut / Copy / Paste	Ctrl + X / Ctrl + C / Ctrl + V
Copy document path	Ctrl + Shift + C
Paste from clipboard history	Ctrl + Shift + V
Duplicate current line or selection	Ctrl + D
Move line up / down	Ctrl + Shift + Up / Down
Delete line at caret	Ctrl + Y
Join / split line	Ctrl + Shift + J / Ctrl + Enter
Start new line	Shift + Enter
Toggle case	Ctrl + Shift + U
Expand / collapse code block	Ctrl + NumPad + / -
Expand / collapse all	Ctrl + Shift + NumPad + / -
Save all	Ctrl + S

VERSION CONTROL

VCS operations popup...	Alt + `
Commit	Ctrl + K
Update project	Ctrl + T
Recent changes	Alt + Shift + C
Revert	Ctrl + Alt + Z
Push...	Ctrl + Shift + K
Next / previous change	Ctrl + Alt + Shift + Down / Up

MASTER YOUR IDE

Find action...	Ctrl + Shift + A
Open a tool window	Alt + [0-9]
Synchronize	Ctrl + Alt + Y
Quick switch scheme...	Ctrl + `
Settings...	Ctrl + Alt + S
Jump to source / navigation bar	F4 / Alt + Home
Jump to last tool window	F12
Hide active / all tool windows	Shift + Esc / Ctrl + Shift + F12
Go to next / previous editor tab	Alt + Right / Alt + Left
Go to editor (from a tool window)	Esc
Close active tab / window	Ctrl + Shift + F4 / Ctrl + F4

FIND EVERYTHING

Search everywhere	Double Shift
Find / replace	Ctrl + F / R
Find in path / Replace in path	Ctrl + Shift + F / R
Next / previous occurence	F3 / Shift + F3
Find word at caret	Ctrl + F3
Go to class / file	Ctrl + N / Ctrl + Shift + N
Go to file member	Ctrl + F12
Go to symbol	Ctrl + Alt + Shift + N

NAVIGATE FROM SYMBOLS

Declaration	Ctrl + B
Type declaration (JavaScript only)	Ctrl + Shift + B
Super method	Ctrl + U
Implementation(s)	Ctrl + Alt + B
Find usages / Find usages in file	Alt + F7 / Ctrl + F7
Highlight usages in file	Ctrl + Shift + F7
Show usages	Ctrl + Alt + F7

REFACTOR AND CLEAN UP

Refactor this...	Ctrl + Alt + Shift + T
Copy... / Move...	F5 / F6
Safe delete...	Alt + Delete
Rename...	Shift + F6
Change signature...	Ctrl + F6
Inline...	Ctrl + Alt + N
Extract method	Ctrl + Alt + M
Introduce variable/ parameter	Ctrl + Alt + V / P
Introduce field / constant	Ctrl + Alt + F / C
Reformat code	Ctrl + Alt + L

ANALYZE AND EXPLORE

Show error description	Ctrl + F1
Next / previous highlighted error	F2 / Shift + F2
Run inspection by name...	Ctrl + Alt + Shift + I
Type/ call hierarchy	Ctrl + H / Ctrl + Alt + H

NAVIGATE IN CONTEXT

Select in...	Alt + F1
Recently viewed / Recent locations	Ctrl + E / Ctrl + Shift + E
Last edit location	Ctrl + Shift + Back
Navigate back / forward	Ctrl + Alt + Left / Right
Go to previous / next method	Alt + Up / Down
Go to line / column...	Ctrl + G
Go to code block end / start	Ctrl +] / [
Add to favorites	Alt + Shift + F
Toggle bookmark	F11
Toggle bookmark with mnemonic	Ctrl + F11
Go to numbered bookmark	Ctrl + [0-9]
Show bookmarks	Shift + F11

BUILD, RUN, AND DEBUG

Run context configuration	Ctrl + Shift + F10
Run / debug selected configuration	Alt + Shift + F10 / F9
Run / debug current configuration	Shift + F10 / F9
Step over / into	F8 / F7
Smart step into	Shift + F7
Step out	Shift + F8
Run to cursor / Force run to cursor	Alt + F9 / Ctrl + Alt + F9
Show execution point	Alt + F10
Evaluate expression...	Alt + F8
Stop	Ctrl + F2
Stop background processes...	Ctrl + Shift + F2
Resume program	F9
Toggle line breakpoint	Ctrl + F8
Toggle temporary line breakpoint	Ctrl + Alt + Shift + F8
Edit / view breakpoint	Ctrl + Shift + F8

jetbrains.com/pycharm
jetbrains.com/help/pycharm
blog.jetbrains.com/pycharm
@pycharm

JET BRAINS

6997

MacOS版：

PyCharm

Find any action inside the IDE	⇧⌘A

CREATE AND EDIT

Show intention actions	⌥↩
Basic / smart code completion	^ Space / ^ ⇧ Space
Type name completion	^⌥Space
Complete statement	⇧⌘↩
Parameter information	⌘P
Quick definition	⌥Space
Quick / external documentation	F1 / ⇧F1
Generate code	⌘N
Override / implement members	^O / ^I
Surround with…	⌥⌘T
Comment with line comment	⌘/
Extend / shrink selection	⌥↑ / ⌥↓
Context info	^⇧Q
Optimize imports	^⌥O
Auto-indent lines	^⌥I
Cut / Copy / Paste	⌘X / ⌘C / ⌘V
Copy document path	⇧⌘C
Paste from clipboard history	⇧⌘V
Duplicate current line or selection	⌘D
Move line up / down	⇧⌘↑ / ⇧⌘↓
Delete line at caret	⌘⌫
Join / split line	^⇧J / ⌘↩
Start new line	⇧↩
Toggle case	⇧⌘U
Expand / collapse code block	⌘+ / ⌘-
Expand / collapse all	⇧⌘+ / ⇧⌘-
Save all	⌘S

VERSION CONTROL

VCS operations popup…	^V
Commit	⌘K
Update project	⌘T
Recent changes	⌥⇧C
Revert	⌥⌘Z
Push…	⇧⌘K
Next / previous change	^⌥⇧↓ / ↑

MASTER YOUR IDE

Find action…	⇧⌘A
Open a tool window	⌘Y [0-9]
Synchronize	⌥⌘Y
Toggle full screen mode	^⌘F
Quick switch scheme…	^`
Preferences…	⌘,
Jump to source / navigation bar	⌘↓ / ⌘↑
Jump to last tool window	F12
Hide active / all tool windows	⇧⎋ / ⇧⌘F12
Go to next / previous editor tab	^Tab
Go to editor (from a tool window)	⎋

FIND EVERYTHING

Search everywhere	Double ⇧
Find / Replace	⌘F / ⌘R
Find in path / Replace in path	⇧⌘F / ⇧⌘R
Next / previous occurence	⌘G / ⇧⌘G
Go to class	⌘O
Go to file member	⌘ F12
Go to file / symbol	⇧ ⌘O / ⌥⌘O

NAVIGATE FROM SYMBOLS

Declaration	⌘B
Type declaration (JavaScript only)	^⇧B
Super method	⌘U
Implementation(s)	⌥⌘B
Find usages / Find usages in file	⌥F7 / ⌘F7
Highlight usages in file	⇧⌘F7
Show usages	⌥⌘F7

REFACTOR AND CLEAN UP

Refactor this…	^T
Copy… / Move…	F5 / F6
Safe delete…	⌘⌫
Rename…	⇧F6
Change signature…	⌘F6
Inline…	⌥⌘N
Extract method	⌥⌘M
Introduce variable / field	⌥⌘V / ⌥⌘F
Introduce constant / parameter	⌥⌘C / ⌥⌘P
Reformat code	⌥⌘L

ANALYZE AND EXPLORE

Show error description	⌘F1
Next / previous highlighted error	F2 / ⇧F2
Run inspection by name…	⌥⇧⌘I
Type / call hierarchy	^H / ^⌥H

NAVIGATE IN CONTEXT

Select in…	⌥F1
Recently viewed / Recent locations	⌘E / ⇧⌘E
Last edit location	⇧⌘⌫
Navigate back / forward	⌘[/ ⌘]
Go to previous / next method	^↑ / ^↓
Go to line /column…	⌘L
Go to code block end / start	⌥⌘] / ⌥⌘[
Add to favorites	⌥⇧F
Toggle bookmark	F3
Toggle bookmark with mnemonic	⌥F3
Go to numbered bookmark	^[0-9]
Show bookmarks	⌘F3

BUILD, RUN, AND DEBUG

Run context configuration	^⌥R
Run / debug selected configuration	^⌥R / ^⌥D
Run / debug current configuration	^R / ^D
Step over / into	F8 / F7
Smart step into	⇧F7
Step out	⇧F8
Run to cursor	⌥F9
Force run to cursor	⌥⌘F9
Show execution point	⌥F10
Evaluate expression…	⌥F8
Stop	⌘F2
Stop background processes…	⇧⌘F2
Resume program	⌥⌘R
Toggle line breakpoint	⌘F8
Toggle temporary line breakpoint	⌥⇧⌘F8
Edit / view breakpoint	⇧⌘F8

jetbrains.com/pycharm
jetbrains.com/help/pycharm
blog.jetbrains.com/pycharm
@pycharm

JET BRAINS

19967

编码规范

编码规范

Guido 的重要见解之一是，代码的读取次数远多于编写次数。

提高代码的可读性并使其在各种 Python 代码中保持一致很重要。总结一句话就是 “可读性很重要”。

PEP 8 是 Python 官方推荐的代码风格指南，旨在提供一致的代码风格，使 Python 代码易于阅读、理解和维护。

`PEP` 是 Python Enhancement Proposal 的缩写，翻译成中文是“Python 增强建议书”，而 8 表示版本。

Python 官方完整 PEP 8 文档地址： https://www.Python.org/dev/peps/pep-0008/

缩进

在 Python 中，缩进是一种非常重要的语法规则，Python 使用缩进来确定代码的层次结构和执行顺序。

- 建议使用 `Tab` 键实现缩进
- 同一级别的代码块的缩进量必须相同

```
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"Name: {self.name}")
        if self.age >= 18:
            print("已成年")
        else:
            print("未成年")
```

注释

注释，是指在代码中对代码功能进行解释的描述性文字，可以提高代码的可读性。注释的内容将被 Python 解释器忽略，并不会在执行结果中体现出来。

Python 中，提供 3 种类型的注释:

- 单行注释 在 Python 中，使用 # 作为单行注释的符号。注释从符号 # 开始直到换行为止，其后面所有的内容都作为注释的内容而被 Python 解释器忽略。

```
# 我是一段注释
```

- 多行注释 在 Python 中，并没有一个单独的多行注释标记，而是将注释内容包含在一对三引号之间，这样的代码将被解释器忽略。由于这样的代码可以分为多行编写，所以也可以作为多行注释。

```
'''
可以使用
三单引号
实现
多行
注释
'''

"""
可以使用
三双引号
实现
多行
注释
"""
```

- 文档注释 文档注释实际是多行注释的一种特殊使用形式，为 Python 文件、模块、类或者函数等添加版权、功能,说明等信息，例如，下面的代码将使用多行注释为程序添加功能、开发者、版权、开发日期等信息，也经常用来解释代码中重要的函数、参数等信息，利于后续开发者维护代码

```
def print(self, *args, sep=' ', end='\n', file=None): # known special case of print
  """
  print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

  Prints the values to a stream, or to sys.stdout by default.
  Optional keyword arguments:
  file:  a file-like object (stream); defaults to the current sys.stdout.
  sep:   string inserted between values, default a space.
  end:   string appended after the last value, default a newline.
  flush: whether to forcibly flush the stream.
  """
  pass
```

命名规范

命名规范在编写代码中起到了很重要的作用，通过使用有意义的命名，可以传达变量、函数和类的用途和含义，使其他人（包括自己）更容易理解代码的意图，避免错误的变量赋值或函数调用。并且当多人合作开发或维护代码时，一致的命名约定使团队成员能够更轻松地理解和修改彼此的代码。

具体包括：

- 包名尽量短小，全小写字母，不推荐使用下划线
- 模块名尽量短小，全小写字母，可以使用下划线分隔多个字母
- 类名采用单词首字母大写形式，即 Pascal 风格。
- 常量命名时全部采用大写字母，可以使用下划线
- 变量、函数名也是全小写字母，多个字母间用下划线 _ 进行分隔
- 使用单下划线 _ 开头的模块变量或者函数是受保护的
- 使用双下划线 __ 开头的实例变量或方法是类私有的

【练习】打印信息

项目简介

打印信息

知识模块

- Python 编程语言

知识点

- 输入和输出

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

分别使用命令行交互模式和 PyCharm 编写程序打印：`测试开发进阶，首选霍格沃兹！`

解题思路

1. 命令行交互模式：

a. 在命令行工具中输入 `python3` ，进入 python 命令行交互方式

b. 输入： `print("测试开发进阶，首选霍格沃兹！")`

c. 回车，查看执行结果

2. PyCharm 中开发：

a. 打开 PyCharm，创建新项目

b. 在项目中新建 python 文件：first.py

c. 在文件中输入：`print("测试开发进阶，首选霍格沃兹！")`

d. 点击鼠标右键，选择 Run 按钮，查看执行结果

完整代码

```
# first.py

print("测试开发进阶，首选霍格沃兹！")
```

代码讲解

1. `print("测试开发进阶，首选霍格沃兹！")`：使用 `print` 函数打印字符串。

1.1.2 Python 基础语法

输入和输出

输入和输出

在编写程序时，输入和输出是非常重要的功能。输入和输出用来获取和显示程序运行所需的数据和程序运行时的信息或结果。

输入函数

在 Python 中，使用 `input()` 函数从键盘获取输入的数据。输入的任何数据，都以字符串形式保存到程序中。

```
name = input()
print("您好, " + name + "!")
```

当程序运行到 `input()` 函数时，程序进入到阻塞状态，等待输入，直至按下回车键。 但是，此时的运行状态并不友好。

`input()` 函数可以在输入数据之前，输出一些提示信息，让输入界面更加友好。

```
name = input("请输入您的姓名：")
print("您好, " + name + "!")
```

输出函数

在 Python 中，可以使用 print() 函数输出文本和变量。

格式：`print(values, sep=" ", end="\n")`

- `values`：需要输出的数据
- `sep=" "`：多个数据之间的分隔符，默认为一个空格
- `end="\n"`：一次输出后的结束符，默认为换行符

基本使用

```
# 输出一个数据
print("Hello World")
# 输出多个数据
print("Hello", "Python", 20, True)
```

指定分隔符

在输出多个数据时，如果不想使用默认的空格做为分隔符，可以通过 `sep` 参数指定分隔符。

```
print("Hello", "Python", 20, True, sep='---')
```

指定结束符

`print` 函数默认一次输出后，都会以换行符结束，下一次输出会重启一个新行输出。 如果在多次输出时，实现在一行输出显示，需要指定结束符。

```
print("Hello", end="")
print("World")
print("Python", end="---")
print("Java")
```

【练习】打印专属用户欢迎信息

项目简介

打印专属用户欢迎信息

知识模块

- Python 编程语言

知识点

- 输入和输出

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

接受用户键盘输入，提示用户输入用户名，打印对应名户名的欢迎信息。

解题思路

1. 在 PyCharm 项目 first.py 文件中编写
2. 获取用户输入：使用 `input()` 函数接受用户输入的用户名，并赋值给变量 `username`
3. 输出多个数据：使用 `print()` 函数输出变量 username 与字符串 `欢迎你~`
4. 点击鼠标右键，选择 Run 按钮，查看执行结果

完整代码

```
# first.py

username = input("请输入用户名：")
print(username, "欢迎你~")
```

代码讲解

1. `username = input("请输入用户名：")`：使用 `input` 函数接受用户输入用户名，然后将其存储在变量 username 中。
2. `print(username, "欢迎你~")`：使用 `print` 函数输入 username 变量与字符串。

函数定义与调用

函数定义与调用

什么是函数

函数是一段可重复使用的代码块，它执行特定的任务并返回一个结果。

在编程中，函数用于将一段代码逻辑封装起来，以便在需要时可以方便地调用。

函数的主要目的是提高代码的重用性和可维护性。

函数通常由以下几个组成部分构成：

1. 函数名：函数名用于标识函数，以便在代码中调用它时使用。
2. 参数：参数是函数在执行时所需的数据。
3. 函数体：函数体是函数的实际代码逻辑，由若干语句组成。
4. 返回值：返回值是函数体代码执行过后的运行结果。

内置函数

内置函数是指开发语言中预先定义好的实现特定功能的函数，可以直接使用。

比如： 输入函数 `input()`，输出函数 `print()` 等

函数定义

虽然系统预先定义好了大量的内置函数，但这些函数，还不能满足实际的开发需求，所以程序中允许根据实际需求自定义函数。

语法格式：

```
def function_name([parameter_list]):
    ['''注释信息''']
    [function_body]
    return [result]
```

说明：

- `def`：自定义函数的关键字。
- `function_name` ： 函数名称，在调用函数时使用，命名需要符合标识符命名规则。
- `()`：函数的特征，用来书写参数列表。
- `parameter_list` ： 用来接收函数运行时所需的外部数据（此章节不讲解）。
- `:`：Python 固定语法。
- `注释信息` ：非必须部分，函数的注释内容，通常是说明该函数的功能、要传递的参数的作用等。
- `function_body`：函数体，用来实现函数功能的逻辑代码。
- `return [result]`：用来结束函数或将函数运行结果返回（此章节不讲解）。

```
def show():
    """
  此函数用来输出Hogwarts
    """
    print("Hogwarts")
```

注意：函数定义完成之后直接运行程序，将不显示任何内容。

函数调用

调用函数也就是执行函数。如果把创建函数理解为理解为创建一个具有某种功能的工具，那么调用函数就相当于使用该工具。

语法格式：

```
function_name([parameters_value])
```

- `function_name`：函数名称，要调用的函数名称，必须是已经创建好的（包括内建的和自定义的）。
- `parameters_value`：参数列表，用来提供函数运行时使用的数据，可省略。
- `()`：函数的特征，参数列表可省略，但圆括号不能省略。

```
print("第一次函数调用")
show()
print("第一次函数调用")
show()
print("程序执行结束")
```

注意：

- 程序在执行到函数调用时，会跳转到函数定义位置执行函数体中的代码。
- 函数体执行结束后，将返回到函数调用处继续向后执行其它代码。

标识符

标识符

什么是标识符

在 Python 中，标识符是用来标识变量、函数、类、模块和其他对象的名称。 需要注意的是，在定义标识符的时候，虽然很自由，但是也不能随心所欲，合法的标识符需要遵从一定的规范。

命名规范

标识符的命名，需要遵从 4 项规范。

1. 见名知意

   一个项目中会存在大量的变量名，函数名，方法名，类名等标识符，所以在对标识符进行命名时，见名知意是第一准则。

   例如：`get_name`、`is_select`、`name`、`age` 等都是符合规则的标识符命名。

   切勿在代码中出现大量 `a1,a2,a3,b1,b2,b3` 的标识符，虽然语法中通过不报错，但在使用时容易混淆。

2. 不能使用关键字

   关键字是 Python 语言中具有特殊含义的单词，例如 `if`、`else`、`for`、`def` 等等。

3. 标识符由英文字母、下划线 _ 和数字组成，但不能以数字开头

   `hogwarts666` 是合法的标识符，而 `666hogwarts` 则是非法的标识符。

4. 区分大小写

   `hogwarts` 和 `Hogwarts` 是两个不同的标识符。

```
hogwarts = 666
Hogwarts = "霍格沃兹"
print(hogwarts)
print(Hogwarts)
```

正确使用标识符是 Python 编程的基础之一。

通过遵守标识符的命名规范，我们可以编写出更加清晰、易于理解的 Python 代码，同时，我们也可以避免一些常见的编程错误。

因此，在编写 Python 代码时，请务必注意标识符的命名规范。

关键字

关键字

什么是关键字

关键字是指在 Python 编程语言中具有特殊含义的保留单词。这些关键字被 Python 解释器用于识别程序的结构和语义，从而执行相应的操作。

在 Python 中，关键字不可以用作变量名、函数名或其他标识符的名称，否则会导致语法错误。

查看关键字

Python 的关键字可能会随着版本的更新而变化，Python 3.9 版本共有关键字 36 个， Python3.10 版本的关键字共有 35 个。

可以通过 Python 内置的 keyword 模块，来查看所有的关键字。

```
# 导入内置关键字模块
import keyword

# 打印所有的关键字
print(keyword.kwlist)
```

执行上面的代码，将会输入以下结果（基于 Python 3.10版本）：

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```


变量

变量

变量的概念

无论使用什么语言编程，总要各种处理数据，比如数字、字符串等等。处理数据就需要使用变量来表示数据。所谓变量，就是用来表示数据的名字。

变量的定义

Python 中定义变量非常简单，只需将数据通过等号(=)赋值给一个符合命名规范的标识符即可。

```
name = "霍格沃兹"
```

Python 是动态强类型语言，变量无须声明类型即可直接赋值，并且变量保存的数据类型可以随时在使用过程中进行改变，Python会根据保存数据的不同对变量类型进行动态推导来确定变量的数据类型。

```
name = "霍格沃兹"
name = 123
```

变量的使用

变量的使用是指在程序中引用一个已经定义的变量。

例如，如果我们想使用之前定义的 `name` 变量，则可以使用：

```
print(name)
```

变量的地址

内置函数 `id()` 可以获取变量的内存地址，也就是这个变量在计算机内存中的唯一标识，通常是用一串数字表示。

每个变量在内存中都有一个唯一的地址，通过比较两个变量的内存地址，可以判断它们是否是同一个变量。如果两个变量的内存地址相同，那么它们就是同一个变量。

```
print(id("霍格沃兹"))

name = "霍格沃兹"
print(id(name))

school = "霍格沃兹"
print(id(school))
```

【练习】函数实现打印信息

项目简介

函数实现打印信息

知识模块

- Python 编程语言

知识点

- 输入和输出
- 函数定义与调用

受众

- 初级测试开发工程师
- 初级 Python 开发工程师

作业要求

定义函数，完成以下功能：提示用户输入编程语言名称，打印信息 `xx是最受欢迎的编程语言！`

解题思路

1. 在 PyCharm 项目中创建 first_func.py 文件
2. 定义函数：`def language()`
3. 获取用户输入：使用 `input()` 函数接受用户输入的编程语言，并赋值给变量 `lan`
4. 输出多个数据：使用 `print()` 函数输出变量 lan 与字符串 `是最受欢迎的编程语言！`
5. 调用函数 `language()`
6. 点击鼠标右键，选择 Run 按钮，查看执行结果

完整代码

```
# first_func.py

def language():
    lan = input("请输入编程语言名称：")
    print(lan, "是最受欢迎的编程语言！")

language()
```

代码讲解

1. `def language():`：定义名为 language 的函数。
2. `lan = input("请输入编程语言名称：")`：函数体，使用 `input` 函数接受用户输入编程语言名称，然后将其存储在变量 lan 中。
3. `print(lan, "是最受欢迎的编程语言！")`：函数体，使用 `print` 函数输入 lan 变量与字符串。
4. `language()`：调用函数 `language` 使其执行。

1.1.3 Python 数据类型

数据类型

数据类型

数据类型是指定程序在运行过程中，将各种数据根据表示形式和组织形式划分为不同的分类。

例如，一个人的姓名可以用字符类型存储，年龄可以用数值类型存储，而婚否可以用布尔类型存储，这些都是 Python 中的基本数据类型。

标准数据类型

Python3 中标准数据类型有：

- 基本数据类型
- Number（数字）
- String（字符串）
- bool（布尔类型）
- 复合数据类型（容器类型）
- List（列表）
- Tuple（元组）
- Dictionary（字典）
- Set（集合）
- 空类型
- None

Python3 根据数据的特性可分为：

- 不可变数据（3 个）：Number（数字）、bool(布尔）、String（字符串）、Tuple（元组）；
- 可变数据（3 个）：List（列表）、Dictionary（字典）、Set（集合）。

类型查看

为了方便检查一个变量的具体类型，可以使用 Python 内置的 `type()` 函数查看变量的类型。

```
# 数字类型
num = 666
print(type(num))

# 布尔类型
b = True
print(type(b))

# 字符串类型
s = "学测试开发升职加薪，就来霍格沃兹测试学社"
print(type(s))

# 空类型
x = None
print(type(x))
```


数字类型

数字类型

Python 中，数字类型（Number) 包括整数(int)、浮点(float)数和复数(complex) 三个子类型。

用来表示程序中不同的数字类型的数据。

整数

整数类型：用来表示整数数值，即没有小数部分的数值，在 Python 中，没有 `Long` 类型，统一使用 `int` 类型表示。

```
n = 10
print(n)
print(type(n))
n = 123123128235470823745082734508237540823745083275408372509823750982735408237504823745082350382475032 45
print(n)
print(type(n))
```

浮点数

浮点类型：用来表示带小数点的数值，在 Python 中，没有 `Double` 类型，统一使用 `float` 类型表示。

```
pi = 3.14
print(pi)
print(type(pi))
pi = 3.14159265234623423434634534634534534534 5
print(pi)
print(type(pi))
```

复数

复数类型：一般在数学和物理学中有广泛的应用，例如在电路分析、信号处理和量子力学中，在此了解即可。

```
n = 4+3j
print(n)
print(type(n))
```


布尔类型

布尔类型

Python 中的布尔类型（bool）是 整数类型（int）的一个子类，用来描述逻辑关系的正确与否。

布尔类型只有两个值，即 `True` 和 `False` 表示真和假，在 Python 中，布尔类型通常用于判断逻辑关系和状态标记。

条件判断

```
x = 5
y = 10
print(x < y)  # 输出True
print(x > y)  # 输出False
```

状态标记

程序开发过程中，经常需要标记一些状态，比如是否选中，是否点击等，都可以使用记录布尔值的变量记录状态。

```
isSelect = False
isExist = True
```

假值状态

在 Python 中，所有的对象都可以判断是否为真。其中，只有下面列出的几种情况得到的值为假，其他对象在 if 或者 while 语句中都表现为真。

常见的假值类型：

- False
- None
- 数值中的零，包括 0、0.0、虚数 0。
- 空序列，包括字符串、空元组、空列表、空字典。

类型转换

类型转换

Python 是动态类型的语言(也称为弱类型语言)，不需要像 Java 或者 C 语言一样必须在使 用变量前声明变量的类型。虽然 Python 不需要先声明变量的类型，但有时仍然需要用到类型转换。

自动转换

自动转换也称为隐式类型转换，程序在执行过程中自动完成，将数据精度低转换为数据精度高的类型。

有些时候，在对不同类型的数据进行运算时，Python 会自动将它们转成同样的类型，然后再计算结果。

```
n = 10 + 3.14
print(n, type(n))

n = False + 1
print(n)
print(type(n))
```

强制转换

强制转换也称为显式类型转换，需要在代码中使用类型函数来转换。

比如通过 input() 函数获取的数字，实际为一个数字字符串，如果需要进行计算，需要将其转换成真正的数字才能进行数据计算。

如果我们不进行类型转换，那么可能会导致程序出现错误或者无法正常运行。

```
# 此代码在执行时会报类型错误
title = "霍格沃兹" + 666
print(title, type(title))

title = "霍格沃兹" + str(666)
print(title, type(title))
```

Python 提供了多种类型转换的函数：

- int(x)：将 x 转换为整数。如果 x 是一个字符串，那么它必须是一个表示整数的字符串，否则将会抛出异常。
- float(x)：将 x 转换为浮点数。如果 x 是一个字符串，那么它必须是一个表示浮点数的字符串，否则将会抛出异常。
- complex(x)：将 x 转换为复数。如果 x 是一个字符串，那么它必须是一个表示复数的字符串，否则将会抛出异常。
- bool(x)：将 x 转换为布尔值。如果 x 是一个空对象、0、空字符串、空列表、空元组、空字典或者 False，那么将返回 False，否则将返回 True。
- str(x)：将 x 转换为字符串。如果 x 是一个对象，那么将会调用该对象的 str()方法，将其转换为字符串。
- chr(x): 将一个整数转换为一个字符，虽然结果是一个字符，但其本质依然是一个字符串类型。

```
# 将字符串转换为整数
x = int("123")
print(x)
print(type(x))

# 将字符串转换为浮点数
y = float("3.14")
print(y)
print(type(y))

# 将数字转换为字符串
z = str(123)
print(z)
print(type(z))

# 将数字转换为布尔类型
b = bool(123)
print(b)
print(type(b))

# 将数字转换为字符
c = chr(65)
print(c)
print(type(c))
```

需要注意的是，如果在强制类型转换时，传递的转换数据不合法导致无法转换，那么将会抛出异常。

```
x = int("霍格沃兹")
```

执行后会提示：ValueError: invalid literal for int() with base 10: '霍格沃兹'。

1.1.4 Python 运算符

运算符

运算符

什么是运算符

运算符是用于进行各种运算操作的符号或关键词。

在数学和计算机编程中，运算符被用来表示不同的运算操作，例如加法、减法、乘法、除法等。

比如:

- `4 + 5`，其中，`4` 和 `5` 为操作数，`+` 为运算符。
- `a = 10`，其中，`a` 和 `10` 为操作数，`=` 为运算符。
- `print()`，其中，`print` 为操作数，`()` 为运算符。

Python 中提供了丰富的运算符，通过这些运算符可以在开过程中实现复杂的逻辑计算。

Python 语言常用运算符如下:

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 成员运算符
- 身份运算符
- 运算符优先级

这些运算符，根据使用操作数的不同，可分为：

- 单目运算符
- 双目运算符
- 三目运算符

在每一门语言中，运算符都是非常简单，但是非常重要，非常难以熟练掌握和灵活应用的知识点。

算术运算符

算术运算符

在编写程序时，可以使用算术运算符来进行基本的数学计算。

Python 中的算术运算符包括加法、减法、乘法、除法、取模和幂运算

运算符	描述
+	加法 ： 两个操作数相加
-	减法 ： 两个操作数相减
*	乘法 ： 两个操作数相乘
/	除法 ： 两个操作数相除，结果为浮点数类型
%	取模 ： 也称为求余运算符， 两个操作相除，取余数为计算结果
**	幂运算 ：返回x的y次幂
//	整除 ： 两个操作数相除，取商，结果为整数类型

加法运算符（+）

加法运算符用于将两个数相加。

例如，a + b 表示将 a 和 b 相加的结果。 如果 a 和 b 都是数字，则加法运算符将执行数学加法操作。 如果 a 和 b 是字符串，则加法运算符将执行字符串连接操作。

```
a = 10
b = 20
c = a + b
print("a + b 的值为：", c)

s1 = "hello"
s2 = "hogwarts"
res = s1 + s2
print("字符串拼接结果为：", res)
```

减法运算符（-）

减法运算符用于将一个数减去另一个数。例如，a - b 表示将 b 从 a 中减去的结果。

```
a = 10
b = 20
c = a - b
print("a - b 的值为：", c)
```

乘法运算符（*）

乘法运算符用于将两个数相乘。例如，a * b 表示将 a 和 b 相乘的结果。

```
a = 10
b = 20
c = a * b
print("a * b 的值为：", c)
```

除法运算符（/）

除法运算符用于将一个数除以另一个数。例如，a / b 表示将 a 除以 b 的结果。需要注意的是，如果除数为 0，将会抛出异常。

在Python中，使用 `/` 进行除法运算时，会得到一个浮点数，如果需要整除运算需要使用 `//` 整除运算符

```
a = 10
b = 20
c = b / a
print("b / a 的值为：", c)
```

取模运算符（%）

取模运算符用于获取两个数相除的余数。例如，a % b 表示将 a 除以 b 的余数。

需要注意的是，取模运算本质上也是除法运算，如果除数为 0，将会抛出异常。

```
a = 10
b = 20
c = b % a
print("b % a 的值为：", c)
```

幂运算符（**）

幂运算符用于获取一个数的指数幂。例如，a ** b 表示将 a 的 b 次方。

```
a = 2
b = 10
c = a ** b
print("a ** b 的值为：", c)
```

整除运算符（//）

整除运算符用于将一个数除以另一个数，该运算符会得到一个整数的商。

需要注意的是，如果除数为 0，也将会抛出异常。

```
a = 10
b = 20
c = b // a
print("b / a 的值为：", c)
```


赋值运算符

赋值运算符

赋值运算符是使用 `=` 做为运算符号，将运算符左侧的数据或表达式的结果，保存到运算符左侧的标识符中。

在使用赋值运算符时，运算符右侧可以是任意类型的数据，但左侧必须是一个变量，否则会报错。

除普通的赋值运算符外，赋值运算符还可以和算术运算符组合成为复合赋值运算符。

Python 中提供的赋值运算符如下表所示：

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

普通赋值运算符（=）

将等号左侧的数据保存到等号右侧的变量中

```
a = 1
s = "Hello"
sum = 1 + 2
```

赋值运算符还支持同时定义多个变量

```
a, b, c = 1, 2, 3
print(a, b, c)
```

复合赋值运算符（+=）

`+=` 运算符是算术运算符 `+` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行加法计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a += 20 # 相当于表达式 a = a + 20
print(a)
```

复合赋值运算符（-=）

`-=` 运算符是算术运算符 `-` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行减法计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a -= 20 # 相当于表达式 a = a - 20
print(a)
```

复合赋值运算符（*=）

`*=` 运算符是算术运算符 `*` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行乘法计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a *= 20 # 相当于表达式 a = a * 20
print(a)
```

复合赋值运算符（/=）

`/=` 运算符是算术运算符 `/` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行除法计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a /= 20 # 相当于表达式 a = a / 20
print(a)
```

复合赋值运算符（//=）

`//=` 运算符是算术运算符 `//` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行整除计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a //= 20 # 相当于表达式 a = a // 20
print(a)
```

复合赋值运算符（%=）

`%=` 运算符是算术运算符 `%` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行取模计算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a %= 20 # 相当于表达式 a = a % 20
print(a)
```

复合赋值运算符（**=）

`**=` 运算符是算术运算符 `**` 与 赋值运算符 `=` 的组合形式，用来简化计算赋值操作。 表达式会将等号左侧变量中的数据与等号右侧的值进行幂运算，然后将计算结果重新保存到等号左侧的变量中。

```
a = 10
a **= 20 # 相当于表达式 a = a ** 20
print(a)
```

复合赋值运算符使用注意

当使用复合赋值运算符时，计算过程上看似对运算符进行展开后运算，而实际执行过程中，复合赋值运算符并不会进行展开操作。

特别是复合赋值运算符和其它运算符一起使用时，要特别注意。

```
n = 2
# 该表达式结果为 14, 并不是10
# 如果一定要展开, 可以理解展开后为 n = n * ( 3 + 4)
n *= 3 + 4
print(n)
```


关系运算符

关系运算符

关系运算符也称为比较运算符，用来对参与运算的两个操作数进行比较，确认两个操作数之间的关系，运算结果会返回一个布尔值

Python 中提供的关系运算符如下表所示：

运算符	描述
==	等于 ： 比较对象是否相等
!=	不等于 ： 比较两个对象是否不相等
>	大于 ： 返回x是否大于y
<	小于 ： 返回x是否小于y
>=	大于等于 ： 返回x是否大于等于y。
<=	小于等于 ： 返回x是否小于等于y。

等于 (==)

`==` 用来判断两个操作数是否相同，如果相同，结果为真 `True` ，如果不同，结果为假 `False` 。

```
print( 1 == 2)
print( 1 == 1)
print( 1 == "2")
print( 2 == "2")
print( "abc" == "abc")
print( "abc" == "ABC")
```

不等于 (!=)

`!=` 用来判断两个操作数是否不同，如果不同，结果为真 `True` ，如果相同，结果为假 `False` 。

```
print( 1 != 2)
print( 1 != 1)
print( 1 != "2")
print( 2 != "2")
print( "abc" != "abc")
print( "abc" != "ABC")
```

大于 (>)

`>` 用来判断左操作数是否大于右操作数，如果大于，结果为真 `True` ，否则，结果为假 `False` 。

```
print( 1 > 2)
print( 1 > 1)
print( 1 > "2")
print( 2 > "2")
print( "abc" > "abc")
print( "abc" > "ABC")
```

小于 (<)

`<` 用来判断左操作数是否小于右操作数，如果小于，结果为真 `True` ，否则，结果为假 `False` 。

```
print( 1 < 2)
print( 1 < 1)
print( 1 < "2")
print( 2 < "2")
print( "abc" < "abc")
print( "abc" < "ABC")
```

大于等于 (>=)

`>=` 用来判断左操作数是否大于或等于右操作数，如果大于或等于，结果为真 `True` ，否则，结果为假 `False` 。

```
print( 1 >= 2)
print( 1 >= 1)
print( 1 >= "2")
print( 2 >= "2")
```

```
print( "abc" >= "abc")
print( "abc" >= "ABC")
```

小于等于 (<=)

`<=` 用来判断左操作数是否小于或等于右操作数，如果小于或等于，结果为真 `True`，否则，结果为假 `False`。

```
print( 1 <= 2)
print( 1 <= 1)
print( 1 <= "2")
print( 2 <= "2")
print( "abc" <= "abc")
print( "abc" <= "ABC")
```


逻辑运算符

逻辑运算符

逻辑运算符一般用来解决当有多个关系条件需要判断时使用，用来确定这些条件组合的方式，运算结果为布尔类型值。

Python 中提供的逻辑运算符如下表所示：

运算符	逻辑表达式	描述
and	x and y	逻辑"与"： 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。
or	x or y	逻辑"或"： 如果 x 是非 0，它返回 x 的计算值，否则它返回 y 的计算值。
not	not x	逻辑"非"： 如果 x 为 True，返回 False。如果 x 为 False，它返回 True。

逻辑与运算符（and）

逻辑与运算符用来连接多个关系条件运算，只有当多个条件同时满足时，结果为真 `True` ，否则为假 `False`

```
print(3 > 2 and 2 > 1)
print(3 < 2 and 2 > 1)
print(3 < 2 and 2 < 1)
print(1 < 2 and "H" + "W")
```

逻辑或运算符（or）

逻辑或运算符用来连接多个关系条件运算，只有当多个条件同时不满足时，结果为假 `False` ，只要其中有一个条件为真，结果即为真 `True`

```
print(3 > 2 or 2 > 1)
print(3 < 2 or 2 > 1)
print(3 < 2 or 2 < 1)
print(1 > 2 and "H" + "w")
```

逻辑非运算符（not）

逻辑非运算符用来对表达式结果进行取反运算，如果表达式结果为真，则取反结果为假 `False` ，如果表达式结果为假，则结果即为真 `True`

```
print(not (3 > 2))
print(not (3 < 2))
```

短路特性

在使用逻辑与运算符 和 逻辑或运算符时，如果自左向右计算结果可以确定整个表达式的结果时，后面的表达式条件便不在计算。

非短路操作

```
result = True and print("Hello, World!1")  # 第一个操作数为True，不能确定后续都为真，所以print语句会执行
print(result)   # 输出 None，print语句的返回值为None

result = False or print("Hello, World!2")  # 第一个操作数为False，不能确定后续都为假，所以print语句会执行
print(result)  # 输出 None，print语句的返回值为None
```

短路操作

```
result = False and 1/0  # 第一个操作数为False，已经可以确认整个表达式的结果，虽然表达式有除0错误，但并不会执行
print(result)  # 结果为False

result = True or 1/0  # 第一个操作数为True，已经可以确认整个表达式的结果，虽然表达式有除0错误，但并不会执行
print(result)  # 结果为True
```

成员运算符

成员运算符

Python 提供了成员运算符，用于判断实例中是否包含了一系列的成员，包括字符串，列表或元组。

如下表所示：

运算符	描述
in	如果在指定的序列中找到值返回 True，否则返回 False。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。

成员运算符 IN

如果被查询成员在目标中存在，结果为真 `True`，如果不存在，结果为假 `False`

```
print("o" in "Hogwarts")
print("K" in "Hogwarts")
print(0 in [1,2,3,4,5])
print(3 in [1,2,3,4,5])
```

成员运算符 NOT IN

如果被查询成员在目标中不存在，结果为真 `True`，如果存在，结果为假 `False`

```
print("o" not in "Hogwarts")
print("K" not in "Hogwarts")
print(0 not in [1,2,3,4,5])
print(3 not in [1,2,3,4,5])
```

身份运算符

身份运算符

Python 中的身份运算符用来判断两个对象的引用是否为同一个，换言之，就是用来比较两个对象的内存地址是否相同。

Python 对数字和字符串做了一些优化以提高性能和减少内存开销。以下是 Python 对数字和字符串做的一些优化：

1. 整数池（Integer Pool）：Python 会在程序运行时预先创建一些整数对象，并将其保存在整数池中。这样，在创建整数对象时，可以重用已存在的对象，而不是创建新的对象。这在一定程度上减少了内存开销。
2. 字符串池（String Pool）：对于较短的字符串，Python 会使用字符串池进行优化。字符串池是一个缓存区，保存着所有共享相同内容的字符串对象。当创建新的字符串对象时，Python 会首先检查是否已经存在相同内容的字符串对象，如果存在，则直接重用已有的对象，避免重复创建新的对象。

因此，在某些情况下，字面量相同的两个对象，实际并不是同一个对象，此时如果需要区分对象的话，就需要使用身份运算符。

身份运算符如下表所示：

运算符	描述
is	is 是判断两个标识符是不是引用自一个对象
is not	is not 是判断两个标识符是不是引用自不同对象

身份运算符 is

`is` 用来判断两个对象内存引用地址是否相同，如果相同，结果为真 `True`，如果不相同，结果为假 `False`

```
# 示例 1
str1 = "Hello"
str2 = "Hello"
print(id(str1))  # 输出第一个字符串对象的内存地址
print(id(str2))  # 输出第二个字符串对象的内存地址
print(str1 == str2)
print(str1 is str2)



# 示例 2
str3 = "Hello, World!" * 1000
str4 = "Hello, World!" * 1000
print(id(str3))  # 输出第一个字符串对象的内存地址
print(id(str4))  # 输出第二个字符串对象的内存地址
print(str3 == str4)
print(str3 is str4)
```

注意：两个字面量相同的对象，内存地址未必相同，就像两个双胞胎，长的相同，但是是两个独立的个体。

身份运算符 is not

`is not` 用来判断两个对象内存引用地址是否不同，如果不同，结果为真 `True`，如果相同，结果为假 `False`

```
# 示例 1
str1 = "Hello"
str2 = "Hello"
print(id(str1))  # 输出第一个字符串对象的内存地址
print(id(str2))  # 输出第二个字符串对象的内存地址
print(str1 == str2)
print(str1 is not str2)



# 示例 2
str3 = "Hello, World!" * 1000
str4 = "Hello, World!" * 1000
print(id(str3))  # 输出第一个字符串对象的内存地址
print(id(str4))  # 输出第二个字符串对象的内存地址
print(str3 == str4)
print(str3 is not str4)
```


is 和 == 的区别

在 Python 中，万物皆对象，而对象的三个基本要素:

- 内存地址
- 数据类型
- 值

而 `is` 与 `==` 都作为常用的判断语句去进行使用，这两者之间的主要区别是：

- `==` 运算符: 只比较两个对象的值，相同返回 `True`，不同返回 `False`。
- `is` 运算符: 比较两个对象的id，相同返回 `True`，不同返回 `False`。

在这种场景下， 两个判断的执行结果均为 `True`。

```
a, b = 1,1
# 判断a, b 是否相等
print(a == b)
print(a is b)
```

在这种场景下， 两个判断的执行结果不一致。

```
a = [1,2,3]
b = [1,2,3]
# 对比值一致, 返回True
print(a == b)
# 对比内存地址不一致, 返回False
print(a is b)
```


三目运算符

三目运算符

三目运算符也称为三元运算符，是指运算符在使用时，需要有三个操作数参与计算。

Python 中也提供三目运算符，但语法上与传统的三目运算符并不相同。

可以将 Python 中的三目运算符理解成是 `if-else` 分支语句的简化单行模式

语法格式： `[on_true] if [expression] else [on_false]`

- `on_true`：条件为真时的结果
- `on_false`：条件为假时的结果
- `expression`：判断条件

Python 会先判断 `expression` 条件表达式的结果，如果条件为真，则结果为 `[on_true]`，条件为假，则结果为 `[on_false]`

```
print("Yes" if True else "No")
print("Yes" if False else "No")
```

实际开发过程中，不建议使用三目运算符，相比较 `if-else` 结构的分支语句，三目运算符的可读性不高。

运算符优先级

运算符优先级

Python中的运算符非常丰富，但在使用过程中除了需要注意各自的特性外，不同的运算符也具有不同的优先级别。

Python 运算符优先级如下表所示：

运算符	描述
(expressions...)	绑定或加圆括号的表达式
[expressions...]， {key: value...}，	列表显示，字典显示，集合显示
x[index]， x[index:index]， x(arguments...)， x.attribute	抽取，切片，调用，属性引用
**	乘方 5
+x， -x	正，负
*， @， /， //， %	乘，矩阵乘，除，整除，取余 6
+， -	加和减
in， not in， is， is not， <， <=， >， >=， !=， ==	比较运算，包括成员检测和标识号检测
not	布尔逻辑非 NOT
and	布尔逻辑与 AND
or	布尔逻辑或 OR
if -- else	条件表达式
=， +=， -=， *=， /=， //=， **=	赋值表达式

注意：因为位运算在后期课程使用过程中使用场景非常少，所以在本课程中并没有讲解（不代表在其它场景中不会使用）。

小技巧：如果在使用运算符的过程中，不能很好的掌握其优先级顺序，最简单的办法就是在需要先计算的表达式上加括号

```
print((2+3)*5)
```

1.1.5 Python 数据结构

字符串

字符串

什么是字符串？

字符串是在任何编程语言中都非常重要的一种数据类型。

在 Python 中，字符串是由引号包裹的任意字符组成的不可变序列，用于表示文本类型数据。

字符串定义

字符串可以通过使用 `单引号` 或 `双引号` 或 `三引号` 来定义，用于表示文本信息，如姓名、消息等。

```
# 使用单引号定义字符串：
name = 'Alice'
# 使用双引号定义字符串：
message = "Hello, world!"
# 使用三引号定义字符串
sql_string = """select * from user where name='tom';"""
```

转义字符

转义字符在字符串中用于表示一些特殊字符或序列，以及插入难以直接输入的字符。

常见的转义字符包括：`\n` 表示换行符，`\t` 表示制表符，`\"` 表示双引号，`\'` 表示单引号，`\\` 表示反斜杠。

通过使用转义字符，可以在字符串中插入特殊字符或表示这些无法直接输入的内容。

```
message = "Hello\nWorld!"
print(message)

# output:
# Hello
# World!
```

字符串下标

下标是指从 `0` 开始的数字编号，也称为索引。

在字符串中，每一个字符都会对应一个下标，通过下标可以获取字符串中该下标对应的字符

语法格式：`字符串对象[下标]`

```
s = "hello"
print(s[0])
print(s[3])
print(s[5]) # 该行代码会报错
```

注意: 下标在使用时，不能大于等于该字符串所有字符的个数，否则会产生下标越界错误。

字符串操作

字符串操作

字符串是在每一门编程语言中都非常重要的数据类型，同时对于字符串也提供了丰富的操作函数。

下面将Python中提供的字符串常用函数进行分类讲解

注意：所有的字符串操作，都不会影响原字符串本身，每次操作后都会得到一个操作后的新字符串对象

统计查找替换类

- `len()` 用来获取参数字符串的字符个数，该函数并不是字符串类型特有的，而是一个通用函数

格式：`len(obj)`

示例：

```python
length = len("Hello")
print(length)
length = len("Hello World")
print(length)
```

- `count()` 返回 `str` 在 `string` 里面出现的次数，如果 `start` 或者 `end` 指定则返回指定范围内 `str` 出现的次数

格式：`count(str, start, end)`

示例：

```python
s = "hello world hello Python"
n = s.count("o")
print(n)
n = s.count("O")
print(n)
n = s.count("or")
print(n)
n = s.count("o",10,30)
print(n)
```

- `index()` 检测 `sub` 是否包含在 `string` 中，如果 `start` 和 `end` 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则抛出一个异常

格式：`index(sub, start, end)`

示例：

```python
s = "Hello"
print(s.index("l"))
print(s.index("l",0,3)) # 区间使用下标位置，左闭右开区间
print(s.index("k"))
```

- `rindex()` 作用同 `index()`,查找子串时从右侧查找，若找不到会抛出一个异常

格式：`rindex(sub, start, end)`

示例：

```python
s = "Hello"
print(s.rindex("l"))
print(s.rindex("l",0,3))
print(s.rindex("k"))
```

- `find()` 检测 `sub` 是否包含在 `string` 中，如果 `start` 和 `end` 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回 -1

格式：`find(sub, start, end)`

示例：

```python
s = "Hello"
print(s.find("l"))
print(s.find("l",0,3))
print(s.find("k"))
```

- `rfind()` 作用同 `find()`,查找子串时从右侧查找，若找不到返回 -1

格式：`rfind(sub, start, end)`

示例：

```python
s = "Hello"
print(s.rfind("l"))
print(s.rfind("l",0,3))
print(s.rfind("k"))
```

- `replace()` 把 string 中的 old 替换成 new,如果 max 指定，则替换不超过 max 次.

格式：`replace(old, new, max)`

示例：

```python
s = "Hello Hello Hello"
print(s.replace("ll","LL"))
print(s.replace("l","L",4))
```

字符串判断类

- `startswith()` 检查字符串是否是以 prefix 开头，是则返回 True，否则返回 False。如果 start 和 end 指定值，则在指定范围内检查.

格式：`startswith(prefix, start, end)`

示例：

```python
url = "https://www.ceshiren.com"
print(url.startswith("https://"))
print(url.startswith("https://", 0, 3))
print(url.startswith("https://", 5, 30))
```

- `endswith()` 检查字符串是否是以 `suffix` 结束，是则返回 `True`，否则返回 `False`。如果 `start` 和 `end` 指定值，则在指定范围内检查.

格式：`endwith(suffix, start, end)`

示例：

```python
url = "https://www.ceshiren.com"
print(url.endswith(".com"))
print(url.endswith(".com", 0, 20))
print(url.endswith(".com", 5, 30))
```

- `isalpha()` 如果 string 至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False

格式：`isalpha()`

示例：

```python
print("abc".isalpha())
print("ABC".isalpha())
print("ABCabc".isalpha())
```

```
print("123".isalpha())
print("a b".isalpha())
print("abc123".isalpha())
print("123abc".isalpha())
print("a@".isalpha())
print("".isalpha())
```

- `isdigit()` 如果 string 只包含数字则返回 True 否则返回 False.

格式：`isdigit()`

示例：

```python
print("123".isdigit())
print("123abc".isdigit())
print("abc123".isdigit())
print("".isdigit())
```

- `isalnum()` 如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False

格式：`isalnum()`

示例：

```python
print("abc".isalnum())
print("ABC".isalnum())
print("ABCabc".isalnum())
print("123".isalnum())
print("abc123".isalnum())
print("123abc".isalnum())
print("a b".isalnum())
print("a@".isalnum())
print("".isalnum())
```

- `isspace()` 如果 string 中只包含空格，则返回 True，否则返回 False.

格式：`isspace()`

示例：

```python
print(" ".isspace())
print("    ".isspace()) # tab键, 由4个空白组成
print("\t".isspace())
print("\n".isspace())
print("\r".isspace())
print("".isspace())
print(" a".isspace())
print("1 ".isspace())
```

- `isupper()` 如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False

格式：`isupper()`

示例：

```python
print("ABC".isupper())
print("ABC123".isupper())
print("123ABC".isupper())
print("A!@#B".isupper())
print("abc".isupper())
print("abC".isupper())
print("abc123".isupper())
print("Abc!@#".isupper())
print("123".isupper())
print("".isupper())
print(" ".isupper())
```

- `islower()` 如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False

格式：`islower()`

示例：

```python
print("abc".islower())
print("abc123".islower())
print("ABC".islower())
print("abC".islower())
print("Abc!@#".islower())
print("123".islower())
print("".islower())
print(" ".islower())
```

- `istitle()` 如果 string 是标题化的（所有单词首字符是大写）则返回 True，否则返回 False

格式：`istitle()`

示例：

```python
print("Username".istitle())
print("User Name".istitle())
print("User_Name".istitle())
print("User.Name".istitle())
print("User+Name".istitle())
print("username".istitle())
print("UserName".istitle())
print("user name".istitle())
print("User name".istitle())
```

字符串转换类

- `capitalize()` 把字符串的第一个字符大写

格式：`capitalize()`

示例：

```python
print("username".capitalize())
print("Username".capitalize())
print("userNAME".capitalize())
print("this is username".capitalize())
```

- `title()` 返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写

格式：`title()`

示例：

```python
print("this is username".title())
print("THIS IS USERNAME".title())
print("tHIS IS username".title())
```

- `upper()` 转换 string 中的小写字母为大写

格式：`upper()`

示例：

```python
print("abc".upper())
print("ABC".upper())
print("abCd".upper())
print("abc123".upper())
print("abc123ABC".upper())
```

- `lower()` 转换 string 中的小写字母为小写

格式：`lower()`

示例：

```python
print("abc".lower())
print("ABC".lower())
print("abCd".lower())
print("abc123".lower())
print("abc123ABC".lower())
```

字符串对齐类

- `center()` 返回一个原字符串居中,并使用空格填充至长度 width 的新字符串，如果指定fillchar参数，则使用指定字符填充，fillchar参数长度只能为1

格式：`center(width, fillchar)`

示例：

```python
print("|"+"hogworts".center(20) + "|")
print("|"+"hogworts".center(5) + "|")
print("|"+"hogworts".center(20, "-") + "|")
```

- `ljust()` 返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串，如果指定fillchar参数，则使用指定字符填充，fillchar参数长度只能为1

格式：`ljust(width, fillchar)`

示例：

```python
print("|"+"hogworts".ljust(20) + "|")
print("|"+"hogworts".ljust(5) + "|")
print("|"+"hogworts".ljust(20, "-") + "|")
```

- `rjust()` 返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串，如果指定fillchar参数，则使用指定字符填充，fillchar参数长度只能为1

格式：`ljust(width, fillchar)`

示例：

```python
print("|"+"hogworts".rjust(20) + "|")
print("|"+"hogworts".rjust(5) + "|")
print("|"+"hogworts".rjust(20, "-") + "|")
```

字符串去除空白类

- `strip()` 删除 string 左右两侧的空白字符, 如果指定chars参数，则删除左右两侧指定的字符

格式：`strip(chars)`

示例：

```python
print("|" + "  hogworts  " + "|")
print("|" + "  hogworts  ".strip() + "|")
print("|" + "  hogworts".strip() + "|")
print("|" + "hogworts  ".strip() + "|")
print("|" + "  h o g w o r t s  ".strip() + "|")
print("|" + "bachogwortsabc".strip("cba") + "|")
```

- `lstrip()` 删除 string 左边的空白字符, 如果指定chars参数，则删除左两侧指定的字符

格式：`lstrip(chars)`

示例：

```python
print("|" + "  hogworts  " + "|")
print("|" + "  hogworts  ".lstrip() + "|")
print("|" + "  hogworts".lstrip() + "|")
print("|" + "hogworts  ".lstrip() + "|")
print("|" + "  h o g w o r t s  ".lstrip() + "|")
print("|" + "bachogwortsabc".lstrip("cba") + "|")
```

- `rstrip()` 删除 string 左边的空白字符, 如果指定chars参数，则删除右两侧指定的字符

格式：`lstrip(chars)`

示例：

```python
print("|" + "  hogworts  " + "|")
print("|" + "  hogworts  ".rstrip() + "|")
print("|" + "  hogworts".rstrip() + "|")
print("|" + "hogworts  ".rstrip() + "|")
print("|" + "  h o g w o r t s  ".rstrip() + "|")
print("|" + "bachogwortsabc".rstrip("cba") + "|")
```

字符串分割类

- `split()` 以 sep 为分隔符分割 string，如果指定 maxsplit 参数，则仅分割 maxsplit次

格式：`split(sep, maxsplit)`

示例：

```python
print("a-b-c-d".split("-"))
print("a-b-c-d".split("-", 2))
print("a--b-c-d".split("-"))
print("a-+b-c-d".split("-+"))
print("a b\tc\nd\re".split())
print("a b c d e".split(" ", 3))
```

- `splitlines()` 使用换行符 `\n` 分割 string，如果指定 `keepends` 参数，则结果中会保留 `\n` 符号

格式：`splitlines(keepends)`

示例：

```python
print("a\nb\nc".splitlines())
print("a\nb\nc".splitlines(True))
```

- `partition()` 从 sep 出现的第一个位置起,把 string 分成一个3元素的元组 (string_pre_sep,sep,string_post_sep),如果 string 中不包含 sep 则 string_pre_str == string，其余元素为空字符串

格式：`parttion(sep)`

示例：

```python
print("This is Hogworts".partition("is"))
print("This is Hogworts".partition("iss"))
```

- `rpartition()` 从右向左 sep 出现的第一个位置起,把 string 分成一个3元素的元组 (string_pre_sep,sep,string_post_sep),如果 string 中不包含 sep 则 string_post_str == string，其余元素为空字符串

格式：`rparttion(sep)`

示例：

```python
print("This is Hogworts".rpartition("is"))
```

```
print("This is Hogworts".rpartition("iss"))
```

字符串连接类

- `+`号 将两个字符串连接生成一个新字符串, `+` 号两侧必须都是字符串

格式：`str1 + str2`

示例：

```python
print("Hello" + "World")
print("Hello" + "123")
print("Hello" + 123)
```

- `*`号 将字符串重复N次后生成一个新字符串

格式：`str * n`

示例：

```python
print("*"* 10)
print("hello"* 10)
```

- `join()` 使用 string 连接可迭代对象中的所有元素，可迭代对象参数中的所有元素必须是字符串

格式：`join(iterable)`

示例：

```python
print("".join(("a","b","c")))
print("-".join(("a","b","c")))
print("->".join(("a","b","c")))
print("->".join(["a","b","c"]))
print("->".join({"a","b","c"}))
print("->".join({"a":"A","b":"B","c":"C"}))
```

编码解码类

- `encode()` 使用 encoding 指定的字符集，对 string 进行编码，转换成二进制字符串

格式：`encode(encoding)`

示例：

```python
print("abc123".encode("gbk"))
print("你好".encode("gbk"))

print("abc123".encode("utf-8"))
print("你好".encode("u8"))
```

- `decode()` 使用 encoding 指定的字符集，对 string 进行解码，转换成字符串对象， string必须是二进制字符串

格式：`decode(encoding)`

示例：

```python
s1 = b'\xc4\xe3\xba\xc3'
s2 = b'\xe4\xbd\xa0\xe5\xa5\xbd'
print(s1.decode("gbk"))
print(s2.decode("utf-8"))

# print(s1.decode("u8"))
# print(s2.decode("gbk"))
```

切片操作

对字符串按指定的范围进行截取，得到一个子字符串，指定范围时，起始下标必须小于结束下标，且子字符串不包含结束下标

格式：`str[start: end: step]`

示例：

```python
s = "abcdefg"

# 普通切片
print(s[0: 2])
# 省略范围
print(s[0:])
print(s[: 2])
print(s[:])
# 指定步长
print(s[::1])
print(s[::2])
# 负下标
print(s[-3: -1])
# 负步长
print(s[-1: -3: -1])
# 逆序
print(s[::-1])
```


【练习】字符串综合实战

项目简介

字符串综合实战

知识模块

- Python 编程语言

知识点

- 字符串操作

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，对一个简单的故事进行如下操作：

- 统计故事中的单词数量。
- 查找主人公的名字在故事中的位置。
- 将主人公的名字替换为你的名字。
- 将故事改写为大写和小写形式。

解题思路

- 使用字符串的方法来处理数据。

完整代码

```
story = "Once upon a time, in a land far away, lived a brave knight named Arthur."

# 统计故事中的单词数量
word_count = len(story.split())
print("单词数量:", word_count)

# 查找主人公的名字在故事中的位置
hero_name = "Arthur"
hero_position = story.find(hero_name)
print("主人公姓名在故事中的位置:", hero_position)

# 将主人公的名字替换为你的名字
your_name = "Alice"
new_story = story.replace(hero_name, your_name)
print("替换名字后:", new_story)

# 将故事改写为大写和小写形式
uppercase_story = story.upper()
lowercase_story = story.lower()
print("大写:", uppercase_story)
print("小写:", lowercase_story)
```

代码讲解

1. `len(story.split())`：使用 `split()` 方法将字符串分割成单词列表，并通过 `len()` 函数获取单词数量。
2. `story.find(hero_name)`:使用 `find()` 方法查找主人公名字在故事中的位置
3. `story.replace(hero_name, your_name)`:使用 `replace()` 方法将主人公名字替换为你的名字
4. `story.upper()`:使用 upper() 方法将故事文本转换为大写形式
5. `story.lower()`:使用 lower() 方法将故事文本转换为小写形式

字符串格式化

字符串格式化

什么是字符串格式化

字符串格式化是一种将变量或值插入到字符串中的过程，以创建具有动态内容的字符串。

在Python中，我们可以使用format()方法和f-string（格式化字符串字面值）来进行字符串格式化。

format()方法

format()方法是一种传统的字符串格式化方法。它使用占位符 {} 来表示要插入的变量或值，并通过传递参数来替换这些占位符。

基本用法

```
name = "Alice"
age = 25
message = "My name is {}, and I am {} years old.".format(name, age)
print(message)
# 输出：My name is Alice, and I am 25 years old.
```

在上述示例中，使用 `{}` 占位符表示要插入的变量或值。在调用 `format()` 方法时，通过参数的顺序将相应的值传递给占位符，从而替换字符串中的占位符部分。最终生成的字符串是根据传入的值进行了相应的替换。

位置参数

`format()` 方法支持使用位置参数来指定变量或值的插入顺序。可以在占位符 `{}` 中使用索引号来指定要插入的参数的位置。

```
name = "Alice"
age = 25
message = "My name is {0}, and I am {1} years old.".format(name, age)
print(message)
# 输出：My name is Alice, and I am 25 years old.
```

在上述示例中，`{0}` 表示第一个参数 `name` 要插入的位置，`{1}` 表示第二个参数 `age` 要插入的位置。

通过在 `format()` 方法中按照顺序传递相应的参数，就可以根据索引号将值插入到指定的位置。最终生成的字符串是根据传入的值和索引号进行了相应的替换。

关键字参数

format()方法还支持使用关键字参数来指定变量或值的插入位置。

```
name = "Alice"
age = 25
message = "My name is {name}, and I am {age} years old.".format(name=name, age=age)
print(message)
# 输出：My name is Alice, and I am 25 years old.
```

在上述示例中，`{name}` 和 `{age}` 分别表示关键字参数 `name` 和 `age` 要插入的位置。通过在 `format()` 方法中使用关键字参数，可以根据参数名将值插入到相应的位置。最终生成的字符串是根据传入的值和参数名进行了相应的替换。

格式化选项

在 `format()` 方法中，还可以使用格式化选项来控制插入值的格式，如字段宽度、对齐方式、小数点精度等。

```
pi = 3.141592653589793
formatted_pi = "The value of pi is approximately {:.2f}".format(pi)
print(formatted_pi)
# 输出：The value of pi is approximately 3.14
```

在上述示例中，{:.2f} 是一个格式化选项，它指定了插入的值应该被格式化为带有两位小数的浮点数。通过在冒号后面指定格式化选项，可以控制插入值的显示方式。在这个例子中，值 pi 被格式化为带有两位小数的浮点数，最终生成的字符串为 "The value of pi is approximately 3.14"。

```
# 字符串对齐
print("The value is ljust: |{:5}|".format("abc"))
print("The value is rjust: |{:<5}|".format("abc"))
print("The value is rjust: |{:>5}|".format("abc"))
# 数字对齐
print("The value is rjust: |{:5}|".format(11))
```

```
print("The value is rjust: |{:<5}|".format(11))
print("The value is rjust: |{:>5}|".format(11))
```

f-string字符串

f-string是Python 3.6及更高版本引入的一种简洁且直观的字符串格式化方法。它使用前缀 `f` 定义字符串，并使用花括号 `{}` 来插入变量或值。

基本用法

```
name = "Alice"
age = 25
message = f"My name is {name}, and I am {age} years old."
print(message)
# 输出：My name is Alice, and I am 25 years old.
```

在上述示例中，字符串前面加上 f 前缀表示这是一个 f-string。在字符串中使用花括号 {} 来插入变量或表达式，它们会在运行时被替换为相应的值。在花括号内部，可以直接使用变量名或表达式，它们会被自动计算并插入到字符串中。最终生成的字符串根据插入的变量或表达式而动态变化。

对齐方式

f-string 的格式化选项方式使用同 format()函数一致

```
# 字符串对齐
print(f"The value is ljust: |{'abc':5}|")
print(f"The value is ljust: |{'abc':<5}|")
print(f"The value is rjust: |{'abc':>5}|")
# 数字对齐
print(f"The value is rjust: |{11:5}|")
print(f"The value is rjust: |{11:>5}|")
print(f"The value is ljust: |{11:<5}|")
# 小数保留
print(f"The value is rjust: |{3.1415926:5.2f}|")
```

表达式和函数调用

在f-string中，可以使用表达式和函数调用来生成动态的字符串内容。

```
name = "Alice"
age = 25
greeting = f"{'Hello' if age < 30 else 'Hi'} {name.upper()}"
print(greeting)
# 输出：HELLO ALICE
```

在上述示例中，使用了一个条件表达式 `{'Hello' if age < 30 else 'Hi'}` 来根据年龄的大小选择不同的问候词。如果年龄小于 `30` ，那么使用 `Hello` ；否则使用 `Hi` 。同时，在插入变量 `name` 后，使用了 `name.upper()` 函数调用来将名称转换为大写字母。

f-string 允许在花括号 `{}` 内部使用任意的表达式和函数调用，这些表达式和函数会在运行时进行求值，并将结果插入到字符串中。这使得我们能够生成更加动态和灵活的字符串内容。

字符串格式化的更多用法

除了简单的变量插入，字符串格式化还支持更多的格式控制选项

- 设置字段宽度和对齐方式
- 控制字段宽度和对齐方式
- 设置小数点精度和浮点数格式化
- 格式化日期和时间
- 格式化数字的千位分隔符
- 使用填充字符进行对齐
- 格式化二进制、十六进制和其他进制数值
- 详细的格式化选项可以参考Python官方文档

总结

- 字符串格式化是将变量或值插入到字符串中的特定位置的过程。
- Python提供了多种字符串格式化的方法，包括format()方法和f-string。
- format()方法使用占位符 {} 进行变量插入。

- f-string使用前缀f定义字符串，并使用花括号 {} 进行变量插入。
- 字符串格式化还支持更多的格式控制选项，例如字段宽度、小数点精度等。
- 通过掌握字符串格式化的方法，你可以在字符串中动态地插入变量或值，创造出灵活且有动态内容的字符串。这将提高你的编程效率并增强代码的可读性。

通过掌握字符串格式化的方法和选项，你可以灵活地处理字符串，并创建具有动态内容的输出。

元组

元组

什么是元组

- 元组是一种数据类型，在Python中用于存储多个元素。元组可以容纳多个值，但它们有一些重要的特点。
- 元组是有序的数据结构，这意味着元组中的元素按照它们的顺序存储，并且可以通过索引进行访问和引用。
- 元组是不可变的，一旦定义后，元组中的数据不可以进行添加，修改和删除等操作。
- 元组是异构的，可以包含不同类型的元素，例如整数、浮点数、字符串等。这使得元组成为一种有效的数据结构，用于存储多种不同类型的元素。

元组的定义

元组的字面量定义

- 元组使用小括号 () 来定义，将元组中的元素括在小括号中。
- 元组中的元素通过逗号，进行分隔，每个元素都可以是不同的数据类型。
- 使用小括号 () 来创建一个空元组。如果没有任何元素需要添加到元组中，空无组没有任何实际意义。
- 定义无组时，逗号是必须的，即使元组只包含一个元素，也需要在元素后面加上逗号，以区分它是一个元组而不是其他数据类型。

```
t1 = (1, 2, "hello", True)
t2 = (1, )
t3 = ()
```

元组的构造方法定义

可以通过元组的构造方法定义元组。 当使用构造方法定义元组时，参数只能是可迭代的对象，构造方法会将参数中的元素构造成为元组的元素。 可以理解为将可迭代的对象强制类型转换为元组。

```
t1 = tuple("abc")
t2 = tuple((1,2,3))
t3 = tuple([1,2,3])
```

元组中元素的引用

元组同字符串一样，也可以使用下标形式引用元组中的元素。并且，下标不能超过元组的元素个数减1，否则会抛出下标越界错误。

```
t = (1,2,3,4,5)
print(t[0])
print(t[3])
print(t[5])
```

元组的切片操作

元组的切片操作同字符串一致

```
t = (1,2,3,4,5)
print(t[0:3])
print(t[:3])
print(t[3:])
print(t[:])
print(t[::-1])
```

元组的特点

- 有序性：元组中的元素按照添加顺序进行存储，并且可以通过索引来访问和引用。这意味着元组中的元素保持其原始顺序，不会发生改变。
- 不可变性：元组的元素是不可修改、删除或添加的。

```
t1 = (1, 2, 3)
t1[0] = 10  # 错误！元组不可修改
```

元组的不可变性使其在某些情况下很有用，例如在需要确保数据的完整性和不变性的场景中。它们也可以用作字典的键或作为函数的参数和返回值。

元组的应用场景

- 存储一组不可变的数据：由于元组的不可变性，它们非常适合存储一组不会发生变化的数据，如常量、配置信息等。您可以使用元组来存储相关的值，以确保数据的完整性和不变性。
- 作为字典的键值对：元组可以作为字典的键值对使用，因为元组是不可变的，可以保证字典中的键的稳定性。相比列表，元组更适合作为字典的键，因为字典的键必须是不可变的。
- 函数返回多个值：函数可以使用元组作为返回值，以便一次性返回多个相关的值。通过返回元组，函数可以将多个数据打包在一起，并且调用函数时可以方便地解包元组，获取其中的各个值。

元组的常用方法

由于元组的不可变特性，所以元组提供的操作方法非常少。

- `len()` 获取元组元素个数 格式： `len(t)`

```
t = (1,2,3,4,5)
print(len(t))
```

- `count()` 统计元组中参数 value 指定值的个数。 格式： `count(value)`

```
t = (1,2,3,4,5,1,2,3,1,2,3,3,3,3)
print(t.count(3))
```

- `index()` 在元组中查找 value 第一次出现的下标。如果指定了范围，则仅在指定范围内查找，如果查找的数据在元组中不存在，会抛出一个错误。格式： `index(value, start,stop)`

```
t = (1,2,3,4,5,1,2,3)
print(t.index(3))
print(t.index(3, 5,10))
```


列表

列表

什么是列表

列表是Python中最常用的数据类型之一。它是一种有序、可变，异构的数据集合，可以存储多个不同类型的元素。

列表的特点

列表是Python中的一种数据结构，具有以下特点：

- 有序性：列表中的元素按照添加的顺序进行存储，每个元素都有一个对应的索引，可以通过索引访问和操作列表中的元素。
- 可变性：列表是可变的，也就是说可以通过索引来修改、删除或插入元素。可以改变列表的长度、内容和顺序。
- 可存储不同类型的元素：列表中可以同时存储不同类型的数据，例如整数、字符串、浮点数、布尔值等。甚至可以存储其他列表或其他复杂的数据结构。

由于列表的有序性、可变性和多样化的数据类型，它是一种非常常用和灵活的数据结构，常用于存储和处理一组相关的数据。列表提供了丰富的方法和操作，使得对数据的管理和处理变得更加方便和高效。

列表的定义

字面量定义

Python 中使用中括号定义列表。

```
l1 = []  # 创建一个空列表
l2 = [1, 2, 3, "hello", True]  # 创建一个包含多个元素的列表
```

由于列表具有可变性，所以空列表的定义是被允许且有意义的。

列表的构造方法定义

可能通过列表的构造方法定义列表。 当使用构造方法定义列表时，参数只能是可迭代的对象，构造方法会将参数中的元素构造成为列表的元素。 可以理解为将可迭代的对象强制类型转换为列表。

```
l1 = list("abc")
l2 = list((1,2,3))
l3 = list([1,2,3])
```

列表中元素的引用

列表同字符串，元组一样，也可以使用下标形式引用列表中的元素。并且，下标不能超过列表的元素个数减1，否则会抛出下标越界错误。

```
l = [1,2,3,4,5]
print(l[0])
print(l[3])
print(l[5])
```

列表中元素的修改

由于列表的可变特性，可以通过下标的方式，对列表中的元素进行修改。

```
l = [1,2,3,4,5]
l[0] = 111
l[3] = 444
```

列表的切片操作

列表的切片操作同字符串，元组一致

```
l = [1,2,3,4,5]
print(l[0:3])
print(l[:3])
print(l[3:])
print(l[:])
print(l[::-1])
```

列表的用途

列表在Python中具有广泛的用途，主要包括以下几个方面：

- 存储一组相关的数据：列表是一种有序的数据结构，可以用于存储一组相关的数据，如学生的成绩、员工的信息、商品的价格等。通过将相关的数据放入列表中，可以方便地进行统一的管理和处理。
- 数据的容器：列表提供了便捷的操作方法，可以进行遍历、搜索、插入和删除等操作。通过索引，可以访问列表中的特定元素；通过遍历，可以逐个处理列表中的元素；通过方法，可以在列表中插入新元素、删除指定元素等。
- 算法和数据结构中的应用：列表是一种重要的数据结构，广泛应用于算法和数据结构的实现中。例如，列表可以用于实现栈（Stack）、队列（Queue）、链表（LinkedList）等数据结构，还可以用于排序算法、搜索算法等的实现。

总之，列表在Python中是一种基础且功能强大的数据结构，用途广泛，可以满足各种不同场景下的数据管理和处理需求。

列表操作

列表操作

由于列表的可变特性,Python为列表提供了丰富的操作方法。

获取列表元素个数

格式：`len(l)`

```python
l = [1,2,3,4,5]
length = len(l)
```

统计查找操作

- `count(value)` 在列表中统计参数 value 出现的次数

```python
l = [1,2,3,4,5,1,2,3,3]
print(l.count(3))
```

- `index(value, start, stop)` 在列表中查找参数 value 第一次出现的下标位置，如果给定范围则只在范围内查找，如果查找目标不存在则抛出错误。

```python
l = [1,2,3,4,5,1,2,3,3]
print(l.index(3))
print(l.index(3,5,10))
```

增加元素

- `append(value)` 向列表最后追加元素

```python
l = []
l.append(1)
print(l)
l.append(1)
print(l)
l.append(2)
print(l)
```

- `extend(iterable)` 将一个可迭代对象的元素依次添加到列表最后

```python
l1 = [1,2,3]
l2 = ["a","b","c"]

l1.append(l2)
print(l1)
l1.extend(l2)
print(l1)
l1.extend("456")
print(l1)
l1.extend(("A","B","C"))
print(l1)
```

- `insert(index, value)` 向列表指定下标位置插入一个元素，原有元素依次后移，如果指定下标超过元素个数，则插入到列表最后。

```python
l = [1,2,3,4,5]

l.insert(0, "A")
print(l)
l.insert(3, "B")
print(l)
l.insert(10, "C")
print(l)
l.insert(9, "D")
print(l)
```

删除元素

- `del` 可以使用 del 关键字结合索引来删除指定位置的元素。如果指定的下标不存在则抛出一个错误。

```
l = [1,2,3,4,5,1,2,3]
del l[0]
print(l)
del l[10]
```

- `remove(value)` 在列表中删除第一个指定的数据，如果删除的数据不存在则抛出错误

```
l = [1,2,3,4,5,1,2,3]
l.remove(3)
print(l)
l.remove(33)
```

- `pop(index)` 从列表中取出并删除指定下标位置的元素，默认取出并删除最后一个元素，如果指定下标不存在，则会抛出一个错误。

```
l = [1,2,3,4,5,1,2,3]
print(l.pop())
print(l)
print(l.pop(3))
print(l)
print(l.pop(10))
```

- `clear()` 清空列表

```
l = [1,2,3,4,5,1,2,3]
l.clear()
print(l)
```

列表排序

- `sort(key, reverse)` 对列表进行排序
- sort方法默认对基本数据类型进行升序排序
- reverse参数将列表元素排序后将列表逆序，实现降序排序
- key 参数用来指定排序规则，比如使用学生的年龄进行排序（此处不讲解该参数的使用，在lambda处讲解）

```
l = ["a","abc","ab","A"]
l.sort()
print(l)
l = ["a","abc","ab","A"]
l.sort(reverse=True)
print(l)
```


元组和列表的区别

元组和列表的区别

相同点

- 元组和列表在Python中，都是有序的，可迭代的数据结构。
- 元组和列表都是异构的，都可以存放不同数据类型的元素。

不同点

- 元组是不可变的，不可以进行增删改操作，一旦定义，无法修改
- 列表是可变的，可以对列表中的元素进行增删改操作，空列表有实际意义。

内存占用

由于元组与列表内部的实现机制不同，在相同元素和个数的情况下，元组占用内存空间更小。

```
from sys import getsizeof
t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
print(getsizeof(t))
print(getsizeof(l))
```

总结对比

对比	元组	列表
定义	(1, 2, 3)	[1, 2, 3]
修改	不支持	支持
添加	不支持	支持
删除	不支持	支持
索引访问	支持	支持
切片	支持	支持
遍历	支持	支持
应用场景	固定的，不会被修改的数据	不固定的，可以被修改的数据
占用内存	较小	较大

【练习】数据统计

项目简介

数据统计

知识模块

- Python 编程语言

知识点

- 列表
- 变量
- 循环语句

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

列表中保存若干个数字，计算数字的和，最小值，最大值，平均数

解题思路

1. 定义一个函数 `func1()` ，在该函数中进行操作。
2. 初始化变量 `n_sum` 为0，用于累计求和。
3. 初始化变量 `n_max` 和 `n_min` 为 `None` ，用于保存最大值和最小值。
4. 创建一个列表 `data` ，其中保存了要处理的数字数据。
5. 使用 `while` 循环遍历列表 `data` 中的每个元素，通过遍历获取每个数字。
6. 在循环内部，依次进行以下操作：
7. 循环结束后，得到数字的和、最大值和最小值。
8. 打印输出数字的和、最大值和最小值。
9. 计算平均数
10. 在代码的其他地方调用函数 `func1()` 来执行，并获取结果。

完整代码

```
def func1():
    n_sum = 0
    n_max = None
    n_min = None
    data = [23456, 234, 23, 423, 423, 423, 423, 412, 3235, 346, 47, 5687, 678, 7, 453623, 4523, 565, 786, 9789, 567,
            34634, 234]
    # for n in data:
    i = 0
    l = len(data)
    while i < l:
        n = data[i]
        n_sum += n
        if n_max == None or n_max < n:
            n_max = n
        if n_min == None or n_min > n:
            n_min = n
        i += 1
    avg = n_sum / n

    print(n_sum)
    print(n_max)
    print(n_min)
    print(avg)
```

代码讲解

1. 首先，代码定义了一个名为 `func1()` 的函数，用于计算数字的和、最小值、最大值和平均数。
2. 在函数内部，定义了变量 `n_sum` ，初始值为0，用于保存数字的和；定义了变量 `n_max` 和 `n_min` ，初始值均为 `None` ，分别用于保存最大值和最小值。
3. 创建了一个列表 `data` ，其中保存了待处理的数字数据。
4. 使用 `while` 循环遍历列表 `data` 中的每个元素。循环使用了索引 `i` 来访问列表中的每个数字。
5. 在循环内部，首先获取当前数字 `n` ，然后执行以下操作： 将当前数字 `n` 累加到变量 `n_sum` 中，即 `n_sum += n` 。 判断当前数字 `n` 是否为最大值： 如果 `n_max` 为 `None` ，表示当前数字是第一个数字，将 `n_max` 更新为当前数字 `n` 。 否则，判断当前数字 `n` 是否大于 `n_max` ，如果是，则将 `n_max` 更新为当前数字 `n` 。 判断当前数字 `n` 是否为最小值： 如果 `n_min` 为 `None` ，表示当前数字是第一个数字，将 `n_min` 更新为当前数字 `n` 。 否则，判断当前数字 `n` 是否小于 `n_min` ，如果是，则将 `n_min` 更新为当前数字 `n` 。 更新索引 `i` ，即 `i += 1` 。
6. 循环结束后，得到了数字的和、最大值和最小值。
7. 接下来，计算平均数 `avg` ，即将数字的和 `n_sum` 除以列表 `data` 的长度。
8. 通过 `print()` 函数分别打印输出数字的和、最大值、最小值和平均数。

字典

字典

字典是Python中的一种容器数据类型，用于存储键值对（key-value）的数据集合。

和现实生活中的字典类似，通过一个 key 对应一个确定唯一的值。

字典是无序的，可变的，且可以存储任意类型的元素

字典的定义

Python 中使用花括号，保存key-value形式表示字典。

key-value 中的 key 必须是一个可哈希的对象，可以使用 hash() 函数来判断数据是否可哈希。 简单理解在一次程序运行结束前，无论该数据什么时候执行hash()函数，都能得到一个唯一确定的值。 一般情况下，不可变对象数据，都可以得到一个哈希值。 所以，理论上，pyhton中的不可变对象数据都可以做为key使用。 但是，为了方便使用，大多数情况下，都会使用**字符串类型**数据做为key使用。

```
# 在一次程序运行过程中, hash函数得会得到同一个哈希值
print(hash("ab"))
print(hash("ab"))
print(hash("ab"))

def myhash():
    print(hash("ab"))

myhash()
```

```
# 可哈希数据
print(hash(123))
print(hash("abc"))
print(hash((1,2,3)))

# 不可哈希数据
print(hash([1,2,3]))
print(hash((1,2,[3])))
```

字典的创建

字面量定义字典

```
d1 = {}
d2 = {"name": "Alice", "age": 25, "gender": "female"}
```

构造方法定义字典

Python 中使用构造方法定义字典的形式很多，了解即可

```
d1 = dict(one=1, two=2, three=3)
d2 = dict([('two', 2), ('one', 1), ('three', 3)])
d3 = dict((('two', 2), ('one', 1), ('three', 3)))
d4 = dict([('two', 2), ['one', 1], ('three', 3)])
d5 = dict({'one': 1, 'two': 2, 'three': 3})
d6 = dict({'one': 1, 'three': 3}, two=2)
d7 = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
```

字典数据访问

字典类型没有类似元组，列表那样的数字下标索引。字典使用 key 来获取对应的值，使用 key的方式同下标索引一样，也使用中括号的方式，将元组，列表中的数字下标索引替换为key。 可以简单将key理解成为自定义的下标索引。

格式： `字典对象[key]`

```
stu = {"name":"Tom", "age": 23, "gender":"male"}
print(stu["name"])
print(stu["age"])
k = "gender"
print(stu[k])
```

字典元素添加与修改

字典中的每一个元素都以键值对形式表示，一个key对应一个value, 在一个字典中，key具有唯一性，当给一个key赋值时，如果key在当前字典中不存在，则是添加数据，如果key存在，则对当前key所对应的值进行修改更新。

格式： `字典对象[key] = value`

```
stu = {"name":"Tom", "age": 23, "gender":"male"}
print(stu)
# 添加新元素
stu["address"] = "BeiJing"
print(stu)
# 修改数据
stu["name"] = "Jack"
stu["address"] = "ShangHai"
print(stu)
```

字典元素的删除

字典也可以使用 `del` 通过key删除元素,当删除元素时，整个键值对都会被删除。

格式： `del 字典对象[key]`

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
print(stu)
# 删除元素
del stu['age']
print(stu)
del stu['address']
print(stu)
```

字典的有序性（Python 3.7+）

在Python 3.7及更高版本中，字典被认为是有序的，即它们可以保持键值对的添加顺序。这意味着当遍历字典或打印字典时，键值对的顺序与它们添加到字典中的顺序相同。这种有序性是字典的内置特性，不需要额外的操作。

然而，在Python 3.6及更早版本中，字典是无序的，无法保持键值对的顺序。这意味着当遍历字典或打印字典时，键值对的顺序是不确定的，可能与它们添加到字典中的顺序不同。

因此，如果在编程中需要依赖字典键值对的顺序，建议使用Python 3.7及更高版本以确保有序性。如果使用旧版本的Python，可以考虑使用collections.OrderedDict来实现有序字典的功能。

字典的应用场景

- 字典适用于存储具有相关性的数据，如用户信息、学生成绩等。每个键值对表示一个独立的数据项，通过键来关联对应的值。
- 字典提供了快速查找和访问数据的能力，通过键可以直接定位对应的值，而不需要遍历整个字典。这使得字典在需要根据特定键快速获取对应值的场景下非常有用。
- 字典作为数据的容器，提供了丰富的操作方法，可以方便地进行遍历、搜索、插入和删除操作。可以通过循环遍历字典的键或值，通过键进行搜索和更新数据，通过键值对的添加和删除来动态修改字典的内容。这种灵活性使得字典成为处理各种数据结构的重要工具。

字典操作

字典操作

字典数据获取类操作

- `keys()` 用来获取字典中所有的 key, 保存到一个列表中，并以 `dict_keys` 类型返回

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
ks = stu.keys()
print(ks)
```

- `values()` 用来获取字典中所有的value, 保存到一个列表中，并以 `dict_values` 类型返回

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
ks = stu.values()
print(ks)
```

- `items()` 用来获取字典中所有的键值对，每一个元素键值对都以一个元组保存，将所有元素元组保存到一个列表中，并以 `dict_items` 类型返回

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
ks = stu.items()
print(ks)
```

- `get(key, default)` 用来获取key对应的值，如果指定的key不存在，则返回默认值。

  字典可以使用 `字典对象[key]` 的形式获取键值对，但是该方法如果指定的 key 不存在，程序会抛出一个错误。此时可以使用 `get()` 替代该取值方法

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
# print(stu["name"])
# print(stu["hobby"])
print(stu.get("name"))
print(stu.get("hobby"))
print(stu.get("hobby","无数据"))
```

字典添加更新类操作

- `setdefault(key,default)` 给一个不存在的key添加一个默认值并将该键值对保存到字典中。

  在一些场景下，字典的key存在，但是该key却没有对应的值，此时，就可以使用该方法，为当前的key添加一个默认值。比如服务端要保存客户端发起请求时携带的请求头中的信息。

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}

stu.setdefault("hobby1")
print(stu)
```


```
stu.setdefault("hobby2", "无")
print(stu)
```

- `fromkeys(keys,val)` 用于创建一个新字典，以序列 keys 中元素做字典的键，value 为字典所有键对应的初始值,默认为 `None`。

  该方法是一个静态方法，需要使用字典类型名 `dict` 调用。 该方法如果给给定 keys 参数，则所有的key对应值都为默认值 `None`,如果给定 val值，则所有key对应的值为 val。

```
ks = ("name", "age", "gender")
s1 = dict.fromkeys(ks)
print(s1)

s2 = dict.fromkeys(ks,"无")
print(s2)
```

- `update(d/iterable)` 使用参数中的数据更新当前字典。

  该方法的参数可以接收一个字典（大多数的使用方式），也可以接收一个可迭代对象，如果参数数据中的key在当前字典中存在，则使用新值更新字典中的键值对，如果参数数据中的key在当前字内中不存在，则将键值对添加到当前字典中。

```
# 更新目标数据是一个字典
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
newStu = {"name":"Jack","hobby":"eat"}
stu.update(newStu)
print(stu)
# 更新目标数据是一个可迭代对象
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
newStu = (("name","Rose"),["hobby","play"])
stu.update(newStu)
print(stu)
```

字典删除类操作

- `popitem()` 用来获取并删除字典中的最后一个键值对，返回一个元组，如果字典为空时，则抛出一个错误

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
v = stu.popitem()
print(v)
print(stu)

v = stu.popitem()
print(v)
print(stu)

print({}.popitem())
```

- `pop(key)` 用于获取并删除字典中指定key对应的键值对。如果指定的key不存在，则抛出错误

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}
v = stu.pop("name")
print(v)
print(stu)
```

- `clear()` 清空字典中所有的键值对元素

```
stu = {'name': 'Tom', 'age': 23, 'gender': 'male', 'address': 'BeiJing'}

print(stu)
stu.clear()
print(stu)
```


集合

集合

什么是集合

- 集合是一种数据类型，用于存储多个元素，并确保元素的唯一性。
- 集合中的元素是无序的，不可通过索引或切片进行访问。
- 集合的主要特点是元素不重复，相同的元素在集合中只会出现一次。
- 我们可以使用大括号 {} 或 set() 函数来定义和创建集合。
- 集合提供了各种集合运算，如并集（两个集合中的所有元素）、交集（两个集合中共有的元素）、差集（第一个集合中存在而第二个集合中不存在的元素）等操作。

集合的创建

不能使用 `{}` 创建一个空集合，因为此种方式创建的类型为字典。

```
# 不能使用花括号 {} 来定义一个空集合
s1 = set()
s2 = {}
print(type(s1))
print(type(s2))

# 使用花括号 {}, 在内部添加元素, 用逗号 , 分隔
my_set = {1, 2, 3, 4, 5}

# 使用内置函数 set() 创建集合
my_set = set([1, 2, 3, 4, 5])

# 集合元素具有唯一性
s = {1,1,1,2,3,4,5,6,6,6,6,6,6,6}
print(s)
```

集合常见的用途包括成员检测、从序列中去除重复项以及数学中的集合类计算，例如交集、并集、差集与对称差集等等。

由于集合不支持下标操作，所以不支持常规方式的获取和修改。

集合操作

集合操作

添加操作

- `add(ele)` 向集合中添加一个元素,如果元素则不产生任何影响

```
s = {1, 2, 3}

s.add(4)
print(s)
s.add("Hello")
s.add("Hello")
print(s)
```

- `update(others)` 更新集合，添加来自 others 中的所有元素,others是一个可迭代对象，如果数据在集合中存在则不更新。

```
s = {1, 2, 3}

s.update((4,5,6))
print(s)
s.update([5,6,7])
print(s)
s.update({6,7,8,9})
print(s)
```

删除操作

- `pop()` 从集合中移除并返回任意一个元素，如果集合为空，则抛出错误

```
s = {1, 2, 3}

print(s.pop())
print(s)
print(s.pop())
print(s)
print(s.pop())
print(s)
print(s.pop())
```

- `remove(elem)` 从集合中移除元素 elem。 如果 elem 不存在于集合中则抛出错误。

```
s = {1, 2, 3}

print(s.remove(1))
print(s)
print(s.remove(3))
print(s)
print(s.remove(5))
```

- `discard(elem)` 如果元素 elem 存在于集合中则将其移除，如果elem不存在，则什么也不做

```
s = {1, 2, 3}

s.discard(1)
print(s)
s.discard(3)
print(s)
s.discard(5)
```

- `clear()` 清空集合

```
s = {1, 2, 3}
s.clear()
print(s)
```

集合数据操作

判断两个集合是否不相交

- `isdisjoint(other)` 如果集合中没有与 other 共有的元素则返回 True。 当且仅当两个集合的交集为空集合时，两者为不相交集合。

```
s = {1, 2, 3}

print(s.isdisjoint({4, 5, 6}))
print(s.isdisjoint({3, 4, 5}))
```

判断集合是否是另一个集合的子集

- `issubset(other)` 检测是否集合中的每个元素都在 other 之中。

```
s = {1, 2, 3}

# 判断 s 是否为 other参数的子集
print(s.issubset({1, 2, 3}))
print(s.issubset({1, 2, 3, 4}))
print(s.issubset({3, 4, 5}))
print("*" * 10)
# 也可以通过运算符 <= 直接判断
print(s <= {1, 2, 3})
print(s <= {1, 2, 3, 4})
print(s <= {3, 4, 5})
print("*" * 10)

# 判断是否为真子集
print(s < {1, 2, 3})
print(s < {1, 2, 3, 4})
print(s < {3, 4, 5})
```

判断集合是否是另一个集合的超集

- `issuperset(other)` 检测是否 other 中的每个元素都在集合之中。

```
s = {1, 2, 3, 4}

# 判断 s 是否为 other参数的超集
print(s.issuperset({1, 2, 3}))
print(s.issuperset({1, 2, 3, 4}))
print(s.issuperset({3, 4, 5}))
print("*" * 10)
# 也可以通过运算符 >= 直接判断
print(s >= {1, 2, 3})
print(s >= {1, 2, 3, 4})
print(s >= {3, 4, 5})
print("*" * 10)

# 判断是否为真超集
print(s > {1, 2, 3})
print(s > {1, 2, 3, 4})
print(s > {3, 4, 5})
```

并集

- `union(*other)` 返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素,other可以指定多个集合。

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
s3 = {5, 6, 7, 8}
print(s1.union(s2))
print(s1.union(s2,s3))
# 也可以使用 | 进行集合并集运算
print(s1 | s2)
print(s1 | s2 | s3)
```

交集

- `intersection(*others)` 返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
s3 = {5, 6, 7, 8}
print(s1.intersection(s2))
print(s1.intersection(s2, s3))
print(s1.intersection(s3))
print("*" * 10)
# 也可以使用 & 进行集合交集运算
print(s1 & s2)
print(s1 & s2 & s3)
print(s1 & s3)
```

差集

- `difference(*others)` 返回一个新集合，包含原集合中在 others 指定的其他集合中不存在的元素。

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
s3 = {5, 6, 7, 8}
print(s1.difference(s2))
print(s1.difference(s2, s3))
```

```
print(s1.difference(s3))
print("*" * 10)
# 也可以使用 - 进行集合差集运算
print(s1 - s2)
print(s1 - s2 - s3)
print(s1 - s3)
```

对称差集

- `symmetric_difference(other)` 返回一个新集合，其中的元素或属于原集合或属于 other 指定的其他集合，但不能同时属于两者。

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
s3 = {5, 6, 7, 8}
print(s1.symmetric_difference(s2))
print(s1.symmetric_difference(s3))
print("*" * 10)
# 也可以使用 ^ 进行集合对称差集运算
print(s1 ^ s2)
print(s1 ^ s3)
```


深拷贝与浅拷贝

深拷贝与浅拷贝

什么是拷贝

- 拷贝是指使用一个已存在一个对象，生成一个新的对象，两个对象在内存中具有独立的存储空间。
- 浅拷贝是指是创建一个新的对象时，只拷贝内容是原始对象的引用，而不是创建原始对象的副本数据。
- 深拷贝是指创建一个新的对象，并递归地复制原始对象及其所有嵌套对象的内容，而不仅仅是复制它们的引用。
- 浅拷贝不具有数据独立性，对象的 `copy()` 方法，`copy` 模块的 `copy()` 方法，工厂方法，切片等方式得到的都是浅拷贝对象。
- 深拷贝具有数据独立性，使用 `copy` 模块中的 `deepcopy()` 方法实现深拷贝。
- 程序的大部分场景都使用浅拷贝。
- 浅拷贝，深拷贝特指容器类型保存的复杂结构，对于基本类型的数据，都是引用指向（不在缓存池中的字符串对像除外）。
- 类似公共排序方法 `sorted()` 实现就可以使用深拷贝，因为该方法返回一个排序后的新列表，该列表可能在程序其它位置被修改，避免影响原列表，深拷贝更适合。

浅拷贝

```
import copy

# 原始数据
originData = [[1,2],{"name":"Tom", "chars":["A","B"]}]

# 使用对象的copy()方法得到浅拷贝对象
copyData1 = originData.copy()
# 使用工厂方法获取浅拷贝对象
copyData2 = list(originData)
# 使用切片方式获取浅拷贝对象
copyData3 = originData[:]
# 使用 copy模块中的copy方法获取浅拷贝对象
copyData4 = copy.copy(originData)

# 拷贝成功的验证，内容相同，地址不同
# 查看所有对象内容
print(originData)
print(copyData1)
print(copyData2)
print(copyData3)
print(copyData4)
# 查看所有对象的址，
print(id(originData))
print(id(copyData1))
print(id(copyData2))
print(id(copyData3))
print(id(copyData4))

# 当修改任意对象时，其它对象都会受影响
copyData3[1]["chars"][1] = "BBB"

# 查看所有对象的数据
print(originData)
print(copyData1)
print(copyData2)
print(copyData3)
print(copyData4)
```

深拷贝

```
import copy

# 原始数据
originData = [[1,2],{"name":"Tom", "chars":["A","B"]}]

# 使用 copy模块中的deepcopy方法获取深拷贝对象
deepCopyData = copy.deepcopy(originData)

# 拷贝成功的验证，内容相同，地址不同
# 查看所有对象内容
print(originData)
print(deepCopyData)

# 查看所有对象的址，
print(id(originData))
print(id(deepCopyData))
```

```
# 当修改任意对象时，其它对象都不会受影响
originData[1]["chars"][1] = "BBB"

# 查看所有对象的数据
print(originData)
print(deepCopyData)
```


1.1.6 Python 流程控制

分支语句 if

分支语句-IF

什么是分支语句

在生活中，总是要做出许多选择，

人有许多的特征（擅长）能决定今后道路的选择，比如说在学习的时候有文科和理科区分，对应的学生们就有擅长/喜欢文科和擅长/喜欢理科的； 针对于不同的擅长/喜欢的方向在将来也有不同的职业发展，擅长/喜欢文科可能以后的择业就会偏向历史、地理等；而擅长/喜欢理科的可能以后的择业方向就有数学、生物、编程（软件）等等方向； 在大方向的选择之后还有更细粒度的选择方向，拿大家学习的软件从业来讲，还有软件测试、软件开发、产品和项目经理等方向可供选择。

程序也是一样。下面给出几个常见的例子：

- 如果购买商品成功，用户余额减少，用户积分增多。
- 如果输入的用户名和密码正确，提示登录成功，进入网站，否则，提示登录失败。
- 如果用户使用微信登录，则使用微信扫一扫；如果使用 QQ 登录，则输入 QQ 号和密码； 如果使用微博登录，则输入微博账号和密码；如果使用手机号登录，则输入手机号和 密码。

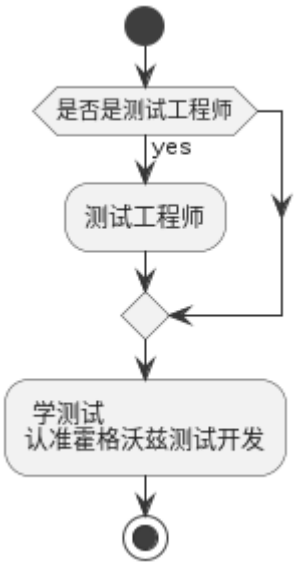
以上例子中的判断，就是程序中的选择语句，也称为条件语句，即按照条件选择执行不同 的代码片段。Python 中选择语句主要有 3 种形式，分别为 if 语句、if…else 语句和 if…elif…else 多分支语句，下面将分别对它们进行详细讲解。

if 条件判断

Python 中使用 if 保留字来组成选择语句，其最简单的语法形式如下：

```
if 表达式:
    语句块
```

其中，表达式可以是一个单纯的布尔值或变量，也可以是比较表达式或逻辑表达式（例如， 是否是测试工程师），如果表达式的值为真，则执行“测试工程师”的打印；如果表达式的值为假，就跳过“语句块”，继续执行后面的语句。这种形式的 if 语句相当于汉语里的“如果……就……”，其流 程图如图所示。



把以上流程转换成为Python代码

```
student = "测试工程师"

if student == "测试工程师":
```

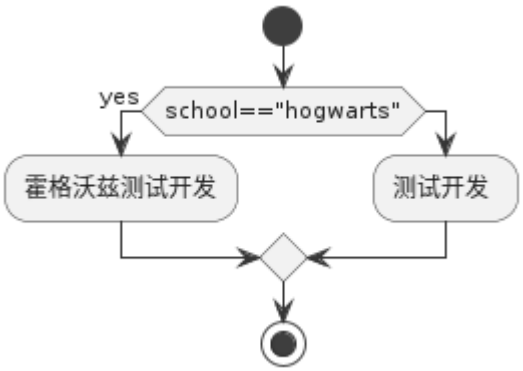
```
    print("测试工程师")
print("学测试\n认准霍格沃兹测试开发")
```

if... else 判断语句

如果遇到只能二选一的条件，Python 中提供了 if…else 语句解决类似问题，其语法格式如下：

```
if 表达式:
    语句块1
else:
    语句块2
```

使用 if…else 语句时，表达式可以是一个单纯的布尔值或变量，也可以是比较表达式或逻辑表达式(例如判断 school 是否为 hogwarts)，如果满足条件，则执行 if 后面的语句块（打印霍格沃兹测试开发），否则，执行 else 后面的语句块（打印测试开发）。这种形式的选择语句相当于汉语里的“如果……否则……”，其流程图如图所示。



把以上流程转换成为Python代码

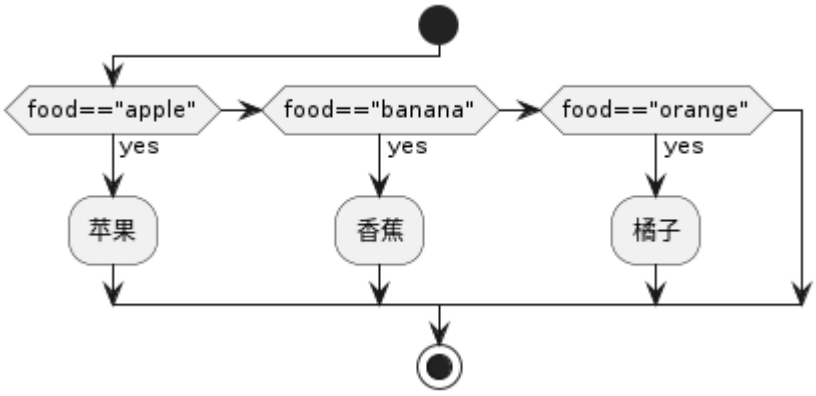
```
student = "测试工程师"
school = "hogwarts"

if school == "hogwarts":
    print("霍格沃兹测试开发")
else:
    print("测试开发")
```

if... elif... else 多重条件

平时网上购物时，通常都有多种付款方式以供选择，比如支付宝支付、银联支付、微信支付等等。这时用户就需要从多个选项中选择一个。在开发程序时，如果遇到多选一的情况，则可以使用 if…elif…else 语句，该语句是一个多分支选择语句，通常表现为“如果满足某种条件，进行某种处理，否则，如果满足另一种条件，则执行另一种处理……”。if…elif…else 语句的语法格式如下：

```
if 表达式:
    语句块1
elif 表达式:
    语句块2
...
else:
    语句块n
```

使用 if…elif…else 语句时，表达式可以是一个单纯的布尔值或变量，也可以是比较表达式或逻辑表达式(例如判断 food 是否为 apple)，如果满足条件，则执行 if 后面的语句块（打印苹果），如果表达式为假，就跳过第一个语句块，进行第二个表达式的执行（例如判断 food 是否为 banana），如果满足条件，则执行 if 后面的语句块（打印香蕉），如果表达式为假，就跳过第二个语句块，进行第三个表达式的执行（例如判断 food 是否为 orange），如果满足条件，则执行 if 后面的语句块（打印橘子），如果表达式为假，就跳过第三个语句块，执行 else 后面的语句块。只有在所有表达式都为假的情况下，才会执行 else 中的语句。这种形式的选择语句相当于汉语里的“如果……如果……否则……”，其流程图如图所示。



把以上流程转换成为 Python 代码

```
food = input("请输入水果的类型:\n")

if food == "apple":
    print("你输入的是苹果")
elif food == "banana":
    print("你输入的是香蕉")
elif food == "orange":
    print("你输入的是橘子")
else:
    print("你触及了我翻译的盲区了:(")
```

分支嵌套

前面介绍了 3 种形式的 if 选择语句，这 3 种形式的选择语句之间都可以进行互相嵌套。例如，在最简单的 if 语句中嵌套 if…else 语句，形式如下：

```
if 表达式 1:
    if 表达式 2:
        语句块 1
    else:
        语句块 2
```

也能在 else 语句中嵌套

```
if 表达式 1:
    if 表达式 2:
        语句块 1
    else:
        语句块 2
else:
    if 表达式 3:
        语句块 3
    else:
        语句块 4
```

还能在 elif 语句中嵌套

```
if 表达式 1:
    if 表达式 2:
        语句块 1
    else:
        语句块 2
elif 表达式 3:
    if 表达式 4:
        语句块 3
    else:
        语句块 4
else:
    if 表达式 5:
        语句块 5
    else:
        语句块 6
```

if 选择语句可以有多种嵌套方式，在开发程序时，可以根据自身需要选择在合适的分支基础上进行嵌套，但一定要严格控制好不同级别代码块的缩进量。

接下来使用本章节所学习的条件语句，将课程开始的例子转换成 Python 代码之后的效果：

```
name = input("请输入你的名字：\n")
hobby = int(input("请选择你擅长/喜欢的科目，文科选1，理科选2：\n"))
if hobby == 1:
    orientation_choose = int(input("请选择你想要的职业，历史选1，地理选2：\n"))
    if orientation_choose == 1:
        orientation = "历史"
    else:
        orientation = "地理"
else:
    orientation_choose = int(input("请选择你想从业的方向：数学选1，生物选2，编程选3\n"))
    if orientation_choose == 1:
        orientation = "数学"
    elif orientation_choose == 2:
        orientation = "生物"
    else:
        coder_choose = int(input("请选择你想从事的软件职业方向：测试选1，开发选2，产品选3，项目经理选4\n"))
        if coder_choose == 1:
            orientation = "测试"
        elif coder_choose == 2:
            orientation = "开发"
        elif coder_choose == 3:
            orientation = "产品"
        else:
            orientation = "项目经理"
print(f"{name}同学，你意向的职业为：{orientation}")
```

通过转换出来的代码，就能使用条件语句 if 实现学生的求职意向调查的简单逻辑了。

【练习】回文数(切片实现)

项目简介

回文数(切片实现)

知识模块

- Python 编程语言

知识点

- 运算符
- 分支语句-if
- 字符串操作

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，输入一个5位数，判断输入的这个数字是否为回文数。回文数是指从左到右和从右到左读都一样的数。例如12321。如果输入的是回文数，输出是回文数，否则输出不是回文数。

解题思路

1. 获取用户输入： 使用 `input()` 函数接收用户输入的一个5位数。
2. 判断是否为回文数： 将输入的数转换为字符串，并逆序排列，然后与原字符串比较。如果两者相等，则为回文数，否则不是回文数。
3. 输出结果： 根据判断结果输出相应的信息。

完整代码

```
num = input("请输入一个5位数：")
if num == num[::-1]:
    print(f"{num}是一个回文数!")
else:
    print(f"{num}不是一个回文数!")
```

代码讲解

1. `num = input("请输入一个5位数：")`：通过 `input` 函数获取用户输入的字符串，并将其赋值给变量 `num`。
2. `if num == num[::-1]`：使用切片 `[::-1]` 反转字符串，然后将反转后的字符串与原始字符串进行比较。如果它们相等，说明这个数是回文数。

【练习】成绩判断

项目简介

成绩判断

知识模块

- Python 编程语言

知识点

- 类型转换
- 分支语句-if

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，用户输入一个分数，程序将根据分数判断并输出相应的等级。分数在 90 分及以上为 A 等级，60 -89 分为 B 等级，否则为 C 等级。

解题思路

1. 获取用户输入的分数。
2. 判断分数所属的等级范围，并根据判断结果输出相应的等级。

完整代码

```
# 用户输入分数
score = int(input('输入分数:\n'))

# 根据分数判断等级
if score >= 90:
    grade = 'A'
elif score >= 60:
    grade = 'B'
else:
    grade = 'C'

# 输出等级和分数
print(f"{score} 属于 {grade} 等级")
```

代码讲解

1. `score = int(input("请输入分数："))`：使用 `input` 函数获取用户输入的分数，并使用 int 函数将输入转换为整数类型，将其存储在变量 `score` 中。
2. `if score >= 90`：使用 `if` 条件语句判断分数是否大于等于90。
3. `elif score >= 60`：使用 `elif` 条件语句判断分数是否大于等于60，同时不满足第一个条件。
4. `else`：如果上述两个条件都不满足，即分数小于60，执行 `else` 语句块。
5. 在每个条件语句块中，将相应的等级（A、B、C）赋值给变量 `grade`。
6. `print(f"{score} 属于 {grade} 等级")`：使用 print 函数输出判断结果，显示用户输入的分数以及对应的等级。

【练习】闰年

项目简介

闰年

知识模块

- Python 编程语言

知识点

- 逻辑运算符
- 三元表达式

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

输入一个4位年份，判断是否是闰年 判断条件：能被4整除但不能被100整除，或能被400整除

解题思路

1. 接收用户输入的年份，并将字符串类型转换为整数类型。
2. 判断条件：年份能被4整除但不能被100整除，或者能被400整除。
3. 使用逻辑运算符组合判断条件，例如使用 `and` 和 `or`。
4. 如果判断条件成立，则该年份是闰年；否则，不是闰年。
5. 打印输出结果。

完整代码

```
year = int(input())
result = "Yes" if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0) else "No"
print(result)
```

代码讲解

1. `year = int(input())`：这一行代码使用 `input()` 函数接收用户输入的年份，并使用 `int()` 函数将其转换为整数类型。`year` 变量将保存用户输入的年份。
2. `(year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)`：这是一个判断条件。根据闰年的定义，如果年份能够被4整除但不能被100整除，或者能够被400整除，那么它就是闰年。这个判断条件使用了逻辑运算符 `and` 和 `or` 来组合两个子条件。
    - `(year % 4 == 0 and year % 100 != 0)`：表示年份能够被4整除但不能被100整除。
    - `(year % 400 == 0)`：表示年份能够被400整除。
1. `result = "Yes" if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0) else "No"`：这是一个使用三元表达式的语句，用于根据判断条件的结果确定年份是否为闰年。如果判断条件成立，则 `result` 变量的值为"Yes"；否则，值为"No"。
2. `print(result)`：使用 `print()` 函数打印输出结果。这里会输出判断结果，即年份是否为闰年的答案。

【练习】计算器

项目简介

计算器

知识模块

- Python 编程语言

知识点

- 类型转换
- 运算符
- 分支语句-if

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个简单的Python程序，实现一个简易的计算器。用户可以输入两个数字和一个运算符（+、-、*、/），程序将根据运算符执行相应的计算操作，并输出结果。

解题思路

1. 提示用户输入要进行的运算操作的运算符。
2. 根据输入的运算符，使用条件语句执行相应的计算操作，并将结果存储在一个变量中。
3. 输出计算结果。

完整代码

```
num1 = float(input("请输入第一个数字："))
num2 = float(input("请输入第二个数字："))
operator = input("请输入运算符（+、-、*、/）：")

# 根据运算符执行相应的计算操作
if operator == '+':
    result = num1 + num2
elif operator == '-':
    result = num1 - num2
elif operator == '*':
    result = num1 * num2
elif operator == '/':
    if num2 != 0:
        result = num1 / num2
    else:
        result = "除数不能为零"
else:
    result = "无效运算符"

# 输出计算结果
print("计算结果：", result)
```

代码讲解

1. `num1 = float(input("请输入第一个数字："))`：这行代码提示用户输入第一个数字，并使用 `input()` 函数获取用户输入的字符串，然后使用 `float()` 函数将其转换为浮点数类型，以便进行数学运算。
2. `num2 = float(input("请输入第二个数字："))`：与上一行类似，这行代码提示用户输入第二个数字并转换为浮点数。
3. `operator = input("请输入运算符（+、-、*、/）：")`：这行代码提示用户输入运算符，运算符可以是加法（+）、减法（-）、乘法（*）或除法（/）。
4. 接下来是条件语句部分，根据用户输入的运算符，执行相应的计算操作。使用 `if`、`elif` 和 `else` 语句来进行条件判断。例如，如果运算符是加法（+），则执行num1 + num2。
5. `print("计算结果：", result)`：使用 `print()` 函数输出计算结果，同时将字符串 "计算结果：" 与结果变量 `result` 进行拼接。

【练习】模拟乘车过程

项目简介

模拟乘车过程

知识模块

- Python 编程语言

知识点

- 分支语句-if

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，模拟乘坐公交车过程，并且可以有座位坐下。要求：输入公交卡当前的余额，只要不低于2元，就可以上公交车；如果车上有空座位，就可以坐下。

解题思路

1. 获取输入的公交卡余额。
2. 判断输入的余额是否大于等于2。
3. 如果余额大于等于2，继续获取车上是否有空座位，再继续进行判断。

完整代码

```
CardCash = int(input('请输入您的公交卡余额：'))
if CardCash >= 2:
    print('余额足够，您可以上车了')
    isSeat = input('车上是否有空座位(有/没有)？')
    if isSeat == '有':
        print('您可以坐下')
    else:
        print('没有座位，只能站着')
else:
    print('余额不够，您不能上车')
```

代码讲解

1. `CardCash = int(input('请输入您的公交卡余额：'))`：使用input() 函数获取用户输入的余额，并转换成 `int` 类型，再赋值给变量 `CardCash`。
2. `if CardCash >= 2`：用于判断余额是否大于等于2。如果条件为真，即余额足够支付车费，继续执行下面的代码。反之，则打印余额不够，您不能上车
3. `isSeat = input('车上是否有空座位(有/没有)？')`：这行代码用于获取用户输入，询问车上是否有空座位，将用户的输入数据赋值给变量 `isSeat`。
4. `if isSeat == '有'`：用于判断是否有空座位，如果条件为真，打印您可以坐下。反之，则打印没有座位，只能站着。

匹配语句 match

匹配语句-MATCH

匹配语句match介绍

在 Python 中，match 是 Python 3.10 版本引入的一种模式匹配语法。

它提供了一种简洁而强大的方式来匹配和处理不同模式的逻辑，当成功匹配一个模式时，就执行该模式后的代码块。

match 语句适用于有限的精准值的匹配，不适用于范围值的判断。

基本语法结构

match 语法的基本结构如下：

```
match value:
    case pattern1:
        # 处理 pattern1 匹配成功的情况
    case pattern2:
        # 处理 pattern2 匹配成功的情况
    ...
    case patternN:
        # 处理 patternN 匹配成功的情况
    case _:
        # 处理所有其他情况的匹配（相当于默认情况）
```

- `match` 表示匹配模式的开始
- `value` 表示需要进行匹配的目标值
- `case` 表示提供一个匹配模式
- `pattern` 表示被匹配的值，该值一般为一个字面量值。
- `_` 表示当所有模式都未匹配成功，则执行该模式下的代码块，本质上 `_` 是一个变量，用来匹配任意值。

示例: 根据输入httpCode值不同，输出对应的描述

```
httpCode = int(input("请输入一个HTTP状态码："))

match httpCode:
    case 101:
        print("临时响应")
    case 200:
        print("请求成功")
    case 301:
        print("重定向")
    case 404:
        print("页面找不到")
    case 500:
        print("服务器内部错误")
    case _:
        print("无效的状态码")
```

`|` 组合多个匹配值

match 可以在一个匹配模式中，提供多个匹配值，使用 `|` 组合，在匹配时，只要成功匹配其中一个值即可。

示例: 改进 HTTPCode 示例，可以匹配不同级别中的更多状态码

```
httpCode = int(input("请输入一个HTTP状态码："))

match httpCode:
    case 100 | 101:
        print("临时响应")
    case 200 | 201 | 203 | 204 | 205:
        print("请求成功")
    case 301 | 304 | 307:
        print("重定向")
    case 401 | 403| 404 | 405:
        print("页面找不到")
    case 500 | 502 | 503:
        print("服务器内部错误")
    case _:
        print("无效的状态码")
```


匹配模式绑定变量

在提供匹配值时，除可以提供字面值外，还可以提供变量，用来匹配任意值，如`_`,但是在一个匹配语句中，只能出现一个独立的匹配变量。

在编写程序过程中，如果需要动态匹配部分数据，比如一个元组中的部分值，此时可以通过绑定变量的方式，提供一个字面值和变量的混合匹配模式，字面值用来精确匹配，变量用来模糊匹配。

示例：输入一个坐标，输出该坐标点的位置。

```
# point is an (x, y) tuple
x = int(input("x:"))
y = int(input("y:"))
point = (x, y)
match point:
    case (0, 0):
        print("坐标在原点上")
    case (0, y):
        print(f"坐标在Y轴上")
    case (x, 0):
        print(f"坐标在X轴上")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("没有这个坐标点")
```

上面代码中：

- 第一个模式有两个字面值，这两个字面值都能精确匹配者能成功匹配当前模式。
- 第二，三两个模式结合了一个字面值和一个变量，而变量绑定了一个来自目标的值（point），此时，字面值需要精确匹配，而绑定的变量可以匹配该位置的任意值。
- 第四个模式捕获了两个值，此时，目标值中的任意值都会被成功匹配。
- 这使得它在概念上类似于解包赋值 (x, y) = point。

循环语句 while

循环语句-WHILE

什么是循环

在日常生活中很多问题都无法一次解决，比如一年中依次按顺序出现春夏秋冬这四个季节，每年季节的变换都重复着春夏秋冬这个过程。有些事物必须周而复始地运转才能保证其存在的意义，例如，公交车、地铁等交通工具必须每天在同样的时间往返于始发站和终点站之间。

类似这样的反复做同一件事的情况，称为循环。

循环主要有两种类型：

- 重复一定次数的循环，称为计次循环，如 for 循环。
- 一直重复，直到条件不满足时才结束的循环，称为条件循环。只要条件为真，这种循环会一直持续下去，如 while 循环。

后面的章节将对这两种类型的循环分别进行介绍。

程序中的循环

- 循环语句允许执行一个语句或语句块多次，当有重复运行一个代码块的需求时，不需要也不能多次复制来实现（有时候事先都无法知道什么时候需要循环结束，所以复制多次来实现基本是不可能的），
- Python 提供了 for 循环和 while 循环两种类型的循环可以应对不同的使用场景。
- 下图是大多数编程语言中循环语句的一般形式

在程序开始之后，会进入一个判断（或者是先执行一个语句后再进入判断），判断条件成立的时候就会再次执行语句块，执行完成该语句块之后会再进行判断条件是否符合，不符合时就会结束循环，否则一直循环执行下去。

说明：在其他语言中（例如，C、C++、Java 等），条件循环还包括 do…while 循环，但 是，在 Python 中没有 do…while 循环。

循环的作用

- 提高代码复用性，减少代码冗余
- 遍历序列（字符串，元组，列表，字典等）

循环的构成要素

- 初始化语句： 一条或多条语句，用于完成一些初始化工作。初始在循环开始之前执行。
- 循环条件：一般是一个布尔表达式，这个表达式能决定是否执行循环体
- 循环体：这个部分是循环的主体，如果循环条件允许，这个代码块将被重复执行。
- 迭代语句：这个部分在每次执行循环体结束后，对循环条件求值之前执行，通常用于控制循环条件中的变量向趋近于结束条件改变。

上面四个部分只是一般分类，并不是每个循环中都非常清晰地分出这几个部分。

什么是 while 循环

在日常的生活中有时候循环的次数通常不会在刚开始就知道，而是满足条件之后就停止循环，如有一路公交车不是固定时间间隔发车，而是在人数满足多少个之后就发车，一直到晚上的11点准时收工，那么没有到11点的话就会一直重复，直到条件不满足时才结束的循环，称为条件循环。只要条件为真，这种循环会一直持续下去，通常指的是 while 循环。

while 循环的语法

```
while 条件表达式:
    循环体
```

当条件表达式的返回值为真时，则执行循环体中的语句，执行完毕后，重新判断条件表达 式的返回值，直到表达式返回的结果为假时，退出循环。

以生活中的例子来理解 while 循环的执行流程，在年会上，主持人要求参与人沿着主持台跑圈。要求当听到主持人喊停止时就停下来。参与者每跑一圈，可能会请求一次主持人发起停止指令。如果老师喊停，则停下来，即循环结束。否则继续跑步，即执行循环。

while 循环实战

实战**1**： 求 1~10 之间所有整数的乘积。

```
# 保存结果的变量
result = 1
# 循环控制变量
n = 2
# 开始循环
while n <= 10:
    # 计算乘积
    result *= n
    # 改变循环变量向结束条件趋近
    n += 1

# 输出结果
print("1~10的乘积为：", result)
```

实战**2**： 输入密码，直到密码输入正确，输出”登录成功“。

```
password = "password" # 设置正确的密码
input_password = ""

while input_password != password:
    input_password = input("请输入密码: ")

print("密码正确, 登录成功！")
```

实战**3**： 在行酒令中，有一个数7小游戏，游戏参与者依次报数，但需要跳过所有包含7或7的整数倍的数字，编写程序找出1~100范围内所有符合该条件的数字。

```
# 循环变量实始化
n = 1
# 循环条件
while n<=100:
    # 数字对7求模为0, 则表示该数字是7的倍数
    # 将数字转换为字符串类型, 使用成员运算符判断字符7是否在字符串中, 检查包含关系
    if n % 7 == 0 or "7" in str(n):
        # 输出满足条件的数字
        print(n)
    # 改变循环变量趋近于结束条件
    n += 1
```


循环语句 for in

循环语句-FOR-IN

Python并没有提供类似C语言中那种传统意义上的for循环，而是提供了一种专门处理字符串，元组，列表，字典等可迭代的序列类型数据的增强型for循环。

for-in 循环的语法

```
for 迭代变量 in 对象:
    循环体
```

for-in 循环会将可迭代对象中的元素依次取出，保存到迭代变量中。每取出一个变量，便执行一次循环体，在循环体中可以通过引用迭代变量，使用取出的数据。

遍历可迭代对象

使用for-in循环处理可迭代对象，可以使操作过程变的极其简单。

遍历字符串

```
s = "Hello Hogworts!"
for c in s:
    print(c)
```

示例： 输出前面示例中每个字符对应的ASCII码值。

```
s = "Hello Hogworts!"
for c in s:
    print(f"字符【{c}】的ASCII码为：【{ord(c)}】")
```

遍历元组

```
t = (1,2,3,4,5)
for n in t:
    print(n)
```

示例：输出前面示例元组中每个数字的立方值

```
t = (1,2,3,4,5)
for n in t:
    print(f"数字【{n}】的立方值为：【{n*3}】")
```

遍历列表

```
requestMethods = ["get", "post", "put","delete", "patch", "header", "options",'trace']
for method in requestMethods:
    print(method)
```

示例：将前面示例列表中所有的请求方式转换为大写输出

```
requestMethods = ["get", "post", "put","delete", "patch", "header", "options",'trace']
for method in requestMethods:
    print(f"请求方式【{method}】转换为大写后：【{method.upper()}】")
```

遍历字典

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for method in requestMethods:
    print(method)
```

从前面的代码中可以看出，字典是一个比较特殊的数据类型，由key-value组成，在使用for-in遍历字典时，默认遍历的是字典的所有key,相当于下面的代码

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
```

```
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for method in requestMethods.keys():
    print(method)
```

如果想遍历字典中的所有值，可以使用下面的方式

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for method in requestMethods.values():
    print(method)
```

但是直接对字典取值遍历的使用方式大多数情况下是无意义的。

示例： 通过遍历字典的key,输出每个key对应的值。

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for method in requestMethods:
    print(f"请求方式【{method}】的作用为：【{requestMethods[method]}】")
```

此种方式虽然可以取到结果，但是for-in循环在遍历字符时，配合字典的 `items()` 方法，实现更简单的方法

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for item in requestMethods.items():
    print(f"请求方式【{item[0]}】的作用为：【{item[1]}】")
```

甚至，代码还可以配合解包操作，更简单的实现

```
requestMethods = {
                    "get": "用于获取服务器上的资源，通过在URL中传递参数来发送请求。",
                    "post": "用于向服务器提交数据，一般用于创建新的资源或进行修改操作。",
                    "put": "用于更新服务器上的资源，一般用于修改已存在的资源的全部内容。",
                    "delete": "用于删除服务器上的资源。"
                }
for key, value in requestMethods.items():
    print(f"请求方式【{key}】的作用为：【{value}】")
```

【练习】回文数(循环实现)

项目简介

回文数(循环实现)

知识模块

- Python 编程语言

知识点

- 类型转换
- 运算符
- 循环语句-for-in

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，输入一个5位数，判断输入的这个数字是否为回文数。回文数是指从左到右和从右到左读都一样的数。例如12321。如果输入的是回文数，输出是回文数，否则输出不是回文数。（使用循环进行实现）

解题思路

1. 获取用户输入： 使用 `input()` 函数接收用户输入的一个5位数。
2. 检查输入的合法性： 确保输入的是一个5位数。如果不是，提醒用户重新输入。
3. 判断是否为回文数： 将输入的数转换为字符串，使用 `for` 循环遍历字符串的一半字符。并逆序排列，然后与原字符串比较。如果两者相等，则为回文数，否则不是回文数。
4. 输出结果： 根据判断结果输出相应的信息。

完整代码

```
a = int(input("请输入一个数字:\n"))
x = str(a)
flag = True

# 遍历字符串的一半字符
for i in range(len(x) // 2):
    if x[i] != x[-i - 1]:
        # 如果字符不相同，将标志变量设为False 并退出循环
        flag = False
        break

if flag:
    print(f"{a} 是一个回文数!")
else:
    print(f"{a} 不是一个回文数!")
```

代码讲解

1. `a = int(input("请输入一个数字:\n"))` ：使用 `input()` 函数获取用户输入的数字。
2. `x = str(a)` ：将输入的数字转换为字符串，方便进行字符比较。
3. 初始化一个标志变量 `flag` 为 `True` ，假设输入的数字是回文数。
4. 使用循环遍历字符串的一半字符， `range(len(x) // 2)` 循环范围是字符串长度的一半，因为只需比较前一半和后一半字符。
5. `x[i] != x[-i - 1]` ：通过比较当前位置和对应位置的字符，判断是否为回文数。
6. 如果字符不相同，将标志变量 `flag` 设为 `False` ，表示输入的数字不是回文数，并立即退出循环。
7. 根据标志变量 `flag` 的值，输出结果，指示输入的数字是否为回文数。

数字序列

数字序列

在使用for-in循环时，默认是无法实现类似传统for循环方式的使用形式，但是可以配合 `range()` 函数实现传统的计次循环。 当然，也可以通过while循环来实现计次循环。 在实现某个功能时，循环可以相互替代实现。

range函数

range() 函数是一个用于生成整数序列的内置函数，通过指定起始值、停止值和步长，它能够生成一个按照指定规则递增或递减的整数序列。

range() 通常用于配合for-in循环结构中控制迭代次数或遍历特定范围的情况使用。

range函数的基本语法

格式： `range(start, stop, step)`

- `start` ：可选参数，表示起始值，默认为0。
- `stop` ：表示结束值，不包含在范围内。
- `step` ：可选参数， 表示步长，默认为1。

```
# 确定开始和结束范围
nums = list(range(1, 10))
print(nums)

# 使用默认开始值
nums = list(range(10))
print(nums)

# 确定范围和步长
nums = list(range(1, 10, 2))
print(nums)

# 使用负步长
nums = list(range(10, 1, -3))
print(nums)
```

for-in结合range函数实现计次循环

```
for i in range(10):
    print(i)
```

示例：计算 1~100 的整数和

```
result = 0
for i in range(1, 101):
    result += i

print(result)
```

随机数

在程序开发过程中，经常会使用到随机数，Python 中 ，可以使用 `random` 模块中的 `randint()` 函数获取随机数。

格式： `randint(start, stop)`

- `start` 为随机数获取初始范围
- `stop` 为随机数获取结束范围，包含该值。
- 使用该函数前需要导入， `from random import randint`

```
from random import randint
print(randint(1, 3))
```

示例： 骰子游戏： 从键盘输入一个数字，和程序随机生成的 1~6范围的数字比较大小

```
from random import randint

play = int(input("请输入一个1-6之间的数字："))
bot = randint(1, 6)

if play == bot:
    print("点数相同，平局")
```

```
elif play > bot:
    print(f"玩家{play}点, 电脑{bot}点, 玩家胜")
else:
    print(f"玩家{play}点, 电脑{bot}点, 电脑胜")
```


【练习】猜数字

项目简介

猜数字游戏

知识模块

- Python 编程语言

知识点

- 随机数生成-random 模块
- 循环语句-while
- 循环语句-for-in
- 用户输入-input()
- 变量操作-统计玩家猜测的总次数

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，实现一个猜数字的游戏。程序随机生成一个1-100目标数字，在一定的范围内，玩家需要根据提示猜测目标数字，直到猜中为止。游戏会根据玩家的猜测给出提示，告诉玩家猜的数字是大了还是小了，最终告诉玩家猜对了，并显示猜测次数。

解题思路

1. 生成目标数字： 使用 `random` 模块随机生成一个目标数字，可以使用 `randint()` 函数来指定数字范围。
2. 接收玩家输入： 使用 `input()` 函数接收玩家猜测的数字。
3. 判断猜测结果： 将玩家输入的猜测数字与目标数字进行比较，如果相等则猜对了，游戏结束。如果猜测数字大于目标数字，给出 "猜大了" 的提示，如果小于目标数字，给出 "猜小了" 的提示。
4. 循环猜测： 使用循环让玩家可以多次猜测，直到猜对为止。在每次循环中，接收玩家输入并判断猜测结果。
5. 记录猜测次数： 在循环中，可以使用一个变量来记录玩家猜测的次数，以便最后告诉玩家猜对时猜了多少次。

完整代码

```
import random

# 生成目标数字
target_number = random.randint(1, 100)

# 初始化猜测次数
guess_count = 0

while True:
    # 接收玩家输入
    guess = int(input("请输入你的猜测数字："))

    # 增加猜测次数
    guess_count += 1

    # 判断猜测结果
    if guess == target_number:
        print(f"恭喜你猜对了！目标数字是 {target_number}，你共猜了 {guess_count} 次。")
        break
    elif guess < target_number:
        print("猜小了，请继续猜测。")
    else:
        print("猜大了，请继续猜测。")
```

代码讲解

1. `import random`：导入 Python 的 `random` 模块，用于生成随机数。
2. `target_number = random.randint(1, 100)`：使用 `random.randint()` 函数生成一个介于 1 到 100 之间的随机目标数字。
3. `guess_count = 0`：初始化猜测次数为 0。
4. `while True`:：进入一个无限循环，使游戏可以持续进行。
5. `guess = int(input("请输入你的猜测数字："))`：接收玩家输入的猜测数字，并将输入的字符串转换为整数。
6. `guess_count += 1`：每次循环猜测次数加一，记录玩家尝试的次数。
7. 判断猜测结果： 如果玩家猜对了（`guess == target_number`），输出恭喜信息，显示目标数字和玩家猜测的次数，然后终止循环。 如果玩家猜测的数字小于目标数字，输出提示让玩家继续猜测。 如果玩家猜测的数字大于目标数字，同样输出提示继续猜测。 游戏继续循环，直到玩家猜对为止。

【练习】猜拳游戏

项目简介

猜拳游戏

知识模块

- Python 编程语言

知识点

- 随机数生成-Python 的内置模块 random
- 分支语句-if-elif-else
- 输入-input()

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实现一个简单的猜拳游戏。玩家和电脑各自选择石头、剪刀或布，根据规则判断胜负关系，并输出比赛结果。

解题思路

1. 导入所需的模块：我们需要使用 random 模块来生成电脑的随机选择。
2. 获取玩家出拳的选项：使用 input 函数提示玩家输入他们的选择，将选择存储在一个变量中。
3. 生成电脑的随机选择：使用 random.choice 函数从一组选项中随机选择电脑的出拳选项，将选择存储在另一个变量中。
4. 输出电脑的出拳选项，以供玩家查看。
5. 使用条件判断来判断胜负关系： 如果玩家和电脑选择相同，游戏结果是平局。 如果玩家选择石头，而电脑选择剪刀，或者玩家选择剪刀，电脑选择布，或者玩家选择布，电脑选择石头，玩家获胜。 否则，电脑获胜。 根据判断结果，输出对应的胜负信息。

完整代码

```
import random

print("猜拳游戏开始！")
player = input("请出拳（石头/剪刀/布）：")
computer = random.choice(["石头", "剪刀", "布"])

print(f"电脑出拳：{computer}")

if player == computer:
    print("平局！")
elif (player == "石头" and computer == "剪刀") or \
     (player == "剪刀" and computer == "布") or \
     (player == "布" and computer == "石头"):
    print("玩家胜利！")
else:
    print("电脑胜利！")
```

代码讲解

1. `import random`：导入 Python 的 `random` 模块，用于生成随机数。
2. `player = input("请出拳（石头/剪刀/布）：")`：使用 input 函数获取玩家的出拳选择，然后将输入的内容存储在变量 `player` 中。
3. `computer = random.choice(["石头", "剪刀", "布"])`：使用 `random.choice` 函数从给定的列表中随机选择一个元素，将结果存储在变量 `computer` 中。
4. `if player == computer:`
   `    print("平局！")`：判断玩家和电脑的选择是否相同，如果相同，输出“平局！”。
5. `elif (player == "石头" and computer == "剪刀") or \`
   `    (player == "剪刀" and computer == "布") or \`
   `    (player == "布" and computer == "石头"):`
   `    print("玩家胜利！")`：使用 `elif` 判断不同的玩家胜利情况。玩家胜利的情况包括：玩家出石头，电脑出剪刀；玩家出剪刀，电脑出布；玩家出布，电脑出石头。如果满足其中任何一个情况，输出“玩家胜利！”。
6. `else:`
   `    print("电脑胜利！")`：如果以上条件都不满足，输出“电脑胜利！”。

【练习】打印图案

项目简介

打印图案

知识模块

- Python 编程语言

知识点

- 循环语句-for-in
- 循环嵌套

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，分别打印以下图案。

图案1正方形如下：

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

图案2直角三角形如下：

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

图案3倒立直角三角形如下：

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

图案4等腰三角形如下：

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

解题思路-正方形

1. 使用嵌套 `for` 循环，外层控制行数，内层控制每行输出的星号数量。
2. 控制换行

完整代码-正方形

```
for i in range(5):
    for j in range(5):
        print("*", end="  ")
    print()
```

代码讲解-正方形

1. 外部循环 for i in range(5): 运行 5 次，总共有5行。
2. 内部循环 for j in range(5): 也运行 5 次，每行输出5个星号。
3. `print("*", end="  ")` - 在内层循环中，使用 `print()` 函数输出一个星号，同时指定 end=" " 参数，将每个星号之间的分隔符设置为两个空格，以保持图案的对齐。
4. `print()` 换行。

解题思路-直角三角形

1. 使用嵌套 `for` 循环，外层控制行数，内层控制每行输出的星号数量。
2. 控制换行

完整代码-直角三角形

```
for i in range(5):
    for j in range(i + 1):
        print("*", end="  ")
    print()
```

代码讲解-直角三角形

1. `for i in range(5)` : - 外层循环迭代变量 i 从 0 到 4，控制行数。总共有 5 行。
2. `for j in range(i + 1)` : - 内层循环迭代变量 j 从 0 到 i，控制每行输出的星号数量。由于每行的星号数量等于当前行数 i 加 1，所以内层循环在每行都会输出相应数量的星号。
3. `print("*", end="  ")` - 在内层循环中，使用 `print()` 函数输出一个星号，同时指定 end=" " 参数，将每个星号之间的分隔符设置为两个空格，以保持图案的对齐。
4. `print()` - 在内层循环结束后，使用一个空的 `print()` 函数来输出一个换行符，以便开始下一行的输出。

解题思路-倒立直角三角形

1. 使用嵌套 `for` 循环，外层控制行数，内层控制每行输出的星号数量。
2. 控制换行

完整代码-倒立直角三角形

```
for i in range(5, 0, -1):
    for j in range(i):
        print("*", end="  ")
    print()
```

代码讲解-倒立直角三角形

1. `for i in range(5, 0, -1)` : 会运行 5 次，从 5 开始递减，每次递减 1，直到达到 1。
2. `for j in range(i)` : 在外部循环的每次迭代中运行 i 次。变量 i 的值对应于每行将要打印的星号数量。
3. `print("*", end="  ")` - 在内层循环中，使用 `print()` 函数输出一个星号，同时指定 end=" " 参数，将每个星号之间的分隔符设置为两个空格，以保持图案的对齐。
4. `print()` 换行，从而为外部循环的下一次迭代创建新的一行。

解题思路-等腰三角形

1. 使用外部循环控制行数，内部循环控制每行的星号和空格的打印。
2. 对于第 i 行，首先在行开始处打印一些空格，以便实现星号的居中对齐。空格数量可以通过行数 i 和三角形总行数的关系来计算。
3. 然后，使用内部循环打印星号。星号的数量是奇数。
4. 打印完星号后，使用换行符 print() 进行换行，为下一行的打印做准备。

完整代码-等腰三角形

```
for i in range(5, 0, -1):
    for j in range(5 - i):
        print(" ", end=" ")
    for j in range(2 * i - 1):
        print("*", end=" ")
    print()
```

代码讲解-等腰三角形

1. `for i in range(1, 5 + 1)`：运行从 1 到 5（包括 5）的循环，控制了等腰三角形的行数。
2. 在每次外部循环的迭代中，使用内部循环 `for j in range(5 - i)`：打印一些空格。这些空格的数量是通过 5 - i 来计算的，以便让星号居中对齐。
3. 然后，内部循环 `for j in range(2 * i - 1)`：打印星号。星号的数量是奇数，通过 `2 * i - 1` 来计算。
4. 使用 `print()` 进行换行，为下一行的打印做准备。

【练习】词频统计

项目简介

词频统计

知识模块

- Python 编程语言

知识点

- 列表
- 列表操作
- 字符串操作
- 字典
- 字典操作
- for-in循环
- 分支语句-if

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，来计算给定文本中每个单词出现的次数。

解题思路

1. 需要将文本转换为小写，这样在统计时不会因为大小写而产生不一致的结果。使用 `.lower()` 方法可以实现。
2. 分割单词： 使用 `.split()` 方法将文本分割成一个单词列表。
3. 词频统计字典： 创建一个空字典来存储每个单词的出现次数。遍历单词列表，去除标点符号，然后在字典中更新每个单词的计数。
4. 循环遍历单词： 使用循环遍历每个单词，可以使用 `.strip(".,!?")` 方法去除单词中的标点符号，以便准确匹配单词。
5. 词频统计更新： 检查单词是否已经在字典中。如果是，增加计数；如果不是，在字典中添加该单词并将计数初始化为1。
6. 输出词频统计结果： 使用 for 循环遍历字典中的键值对，并使用格式化字符串输出每个单词和其出现次数。

完整代码

```
text = """
Python is a popular programming language. It is widely used for web development, data science, and more.
Python has a simple and readable syntax, which makes it great for beginners.
"""
words = text.lower().split()  # 将文本转换为小写并分割为单词
word_count = {}  # 用于存储单词和出现次数的字典

for word in words:
    word = word.strip(".,!?")  # 去除单词中的标点符号
    if word in word_count:
        word_count[word] += 1
    else:
        word_count[word] = 1

# 输出词频统计结果
for word, count in word_count.items():
    print(f"{word}: {count}")
```


代码讲解

1. `words = text.lower().split()`：这行代码使用 `lower()` 方法将文本转换为小写形式，然后使用 `split()` 方法将文本分割为单词列表，并将结果存储在变量 `words` 中。
2. `word_count = {}`：创建一个空字典 `word_count`，用于存储每个单词以及其出现次数。
3. ```python
   for word in words:
   word = word.strip(".,!?")  # 去除单词中的标点符号
   if word in word_count:
       word_count[word] += 1
   else:
       word_count[word] = 1
   ```
   - 这是一个循环，遍历 `words` 列表中的每个单词。
   - 使用 `strip(".,!?")` 方法去除单词中可能的标点符号，以便准确匹配单词。
   - 使用条件语句判断单词是否已经在 `word_count` 字典中。如果在字典中，增加计数；如果不在字典中，将单词添加到字典并初始化计数为1。
4. ```python
   for word, count in word_count.items():
       print(f"{word}: {count}")
   ```
   - 用于遍历 `word_count` 字典中的键值对。
   - 使用格式化字符串输出每个单词和其出现次数。

【练习】水仙花数

项目简介

水仙花数

知识模块

- Python 编程语言

知识点

- 运算符
- 循环语句-for-in
- 分支语句-if
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，找出100-999范围内的水仙花数。所谓"水仙花数"是指一个三位数，其各位数字立方和等于该数本身。例如：153是一个"水仙花数"，因为153=1的三次方＋5的三次方＋3的三次方。

解题思路

1. 循环遍历100-999之间的所有三位数
2. 分别获取百位数，十位数，个位数
3. 判断是否为水仙花数
4. 打印符合条件的数字

完整代码

```
for n in range(100, 1000):
    # 获取百位数
    i = n // 100

    # 获取十位数
    j = n // 10 % 10

    # 获取个位数
    k = n % 10

    # 判断是否为水仙花数
    if n == i**3 + j**3 + k**3:
         # 输出水仙花数
        print(n)
```

代码讲解

1. `for n in range(100, 1000)`：使用 `for` 循环遍历从 100 到 999 的所有三位数。
2. `i = n // 100`：获取数字 n 的百位数。
3. `j = n // 10 % 10`：获取数字 n 的十位数。
4. `k = n % 10`：获取数字 n 的个位数。
5. `if n == i**3 + j**3 + k**3`：判断是否为水仙花数，即判断条件为原数字等于各位数字的立方和。
6. `print(n)` ：如果是水仙花数，将其输出。

【练习】快速排序

项目简介

快速排序

知识模块

- Python 编程语言

知识点

- 递归
- 分区操作
- 列表操作
- 算法复杂度分析

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实现快速排序。

解题思路

1. 首先选择一个基准元素，通常是选择待排序列表的第一个元素。
2. 将列表分成两部分，小于基准元素的元素放在左边，大于基准元素的元素放在右边。这个过程称为分区（partition）。
3. 递归地对左右两个分区进行快速排序。也就是将左边的分区作为新的子列表，再次选择基准元素，并进行分区操作。对右边的分区也是同样的操作。
4. 递归的结束条件是分区中只有一个元素或为空，此时分区已经是有序的。
5. 最后将所有的分区合并起来，即可得到排好序的列表。

完整代码

```
def quickSort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]  # 选取数组中间元素作为基准值
    left = [x for x in arr if x < pivot]  # 小于基准值的部分放在左边列表
    middle = [x for x in arr if x == pivot]  # 等于基准值的部分放在中间列表
    right = [x for x in arr if x > pivot]  # 大于基准值的部分放在右边列表

    return quickSort(left) + middle + quickSort(right)  # 合并结果返回
```

代码讲解

1. `def quickSort(arr):` 定义了一个名为quickSort的函数，它接受一个参数arr，即待排序的列表。
2. `if len(arr) <= 1:` 检查列表的长度是否小于等于1，如果是，则直接返回列表本身，因为长度小于等于1的列表已经是有序的。
3. `pivot = arr[len(arr) // 2]` 选择列表的中间元素作为基准值。使用//是为了得到整数结果，确保索引取到中间元素。
4. `left = [x for x in arr if x < pivot]` 使用列表推导式创建了一个新列表left，其中包含所有小于基准值的元素。
5. `middle = [x for x in arr if x == pivot]` 使用列表推导式创建了一个新列表middle，其中包含所有等于基准值的元素。
6. `right = [x for x in arr if x > pivot]` 使用列表推导式创建了一个新列表right，其中包含所有大于基准值的元素。
7. `return quickSort(left) + middle + quickSort(right)` 通过递归调用quickSort函数分别对left列表和right列表进行排序，并将排序好的left列表、middle列表和right列表拼接成完整的有序列表，然后返回该列表作为函数的结果。

【练习】冒泡排序

项目简介

冒泡排序

知识模块

- Python 编程语言

知识点

- 参数说明
- 条件语句 if-else
- 循环嵌套
- 列表操作
- 判断

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实现冒泡排序。

解题思路

1. 获取待排序列表的长度，用于确定外层循环的次数。
2. 外层循环：从第一个元素开始，遍历到倒数第二个元素，每次循环找到剩余元素中的最大值。
3. 内层循环：在每次外层循环时，从第一个元素开始，遍历到倒数第 `i+1` 个元素，比较相邻的两个元素的大小。
4. 根据是否指定了比较键 `key` 来确定比较的方式。
5. 继续进行内层循环，直到所有的元素都经过比较和交换。
6. 完成外层循环后，列表已经按照指定的顺序进行了排序。
7. 根据 `reverse` 的值确定返回的排序结果：

完整代码

```
def bubbleSort(obj, key=None, reverse=False):
    length = len(obj)
    # 外层循环用来确定找出剩余数据的最大值，N个数，找n-1次即可
    for i in range(length-1):
        for j in range(length-i-1):
            if key:  # if key != None:
                if key(obj[j]) > key(obj[j + 1]):
                    obj[j], obj[j + 1] = obj[j + 1], obj[j]
            else:
                if obj[j] > obj[j+1]:
                    obj[j], obj[j+1] = obj[j+1], obj[j]
    # return obj if not reverse else obj[::-1]
    if reverse:
        return obj[::-1]
    else:
        return obj
```

代码讲解

1. 首先定义了一个名为 `bubbleSort` 的函数，用于实现冒泡排序算法。
2. `bubbleSort` 函数接受三个参数：`obj`、`key` 和 `reverse`。其中 `obj` 是待排序的列表，`key` 是一个可选的函数参数，用于指定比较的键，`reverse` 则是一个可选的布尔参数，用于指定排序的顺序。
3. 在函数中，使用 `len(obj)` 获取待排序列表的长度，即列表中元素的个数。
4. 然后使用外层循环进行迭代，循环次数为数组长度减1。每次循环将找出剩余数据的最大值放在最后的位置。
5. 在外层循环中，使用内层循环从第一个元素开始遍历，直到倒数第 `i+1` 个元素。通过比较相邻的两个元素的大小，如果前一个元素比后一个元素大，则交换它们的位置，确保较大的元素在后面。
6. 在两个嵌套的循环中，通过判断 `key` 是否存在，来决定使用哪种比较方式。如果指定了 `key`，则使用 `key(obj[j])` 和 `key(obj[j + 1])` 进行比较；否则直接使用元素本身进行比较。
7. 最后，根据 `reverse` 参数的值确定返回的排序结果。如果 `reverse` 为 `True`，则将列表反转后返回，即返回逆向排序的结果，即降序；如果未指定或 `reverse` 为 `False`，则直接返回列表，即返回正向排序的结果，即升序。

【练习】选择排序

项目简介

选择排序

知识模块

- Python 编程语言

知识点

- 列表操作
- 循环嵌套
- 判断
- 时间复杂度分析
- 变量
- 索引

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实选择速排序。

解题思路

1. 首先，设置一个变量min_index用于记录当前循环中最小元素的索引。
2. 外层循环从列表的第一个元素开始，依次遍历到倒数第二个元素。
3. 内层循环从外层循环的下一个元素开始，依次遍历到最后一个元素。
4. 在内层循环中，比较当前元素和最小元素，如果当前元素比最小元素小，则更新min_index为当前元素的索引。
5. 内层循环结束后，将最小元素与外层循环的当前元素交换位置。
6. 外层循环继续，重复步骤2-5，直到遍历到倒数第二个元素。
7. 排序完成后，列表中的元素将按照从小到大的顺序排列。

完整代码

```
def xuanze():
    data = [1,56,6,5,4,89,2]
    length = len(data)
    for i in range(length-1):
        small = data[i]
        for j in range(i+1,length):
            if data[j]<small:
                idx = j
                data[idx],data[i] = data[i],data[idx]
    return data

result = xuanze()
print(result)
```

代码讲解

1. 首先，我们定义了一个函数 `xuanze()` ，函数内部创建了一个列表 `data` 并初始化了一些数据。然后，我们获取了列表的长度，用变量 `length` 保存。
2. 接下来，我们使用两层循环实现选择排序的逻辑。外层循环从第一个元素遍历到倒数第二个元素，表示每次选择排序的起始位置。
3. 内层循环从外层循环的下一个位置开始，依次遍历到列表的最后一个元素。内层循环的作用是找到未排序部分的最小值，并将其与当前起始位置的元素交换位置。
4. 在内层循环中，我们使用变量 `small` 来记录当前未排序部分的最小值，并使用变量 `idx` 来记录最小值的索引位置。
5. 如果内层循环中遇到比 `small` 更小的元素，就更新 `idx` 的值为该元素的索引。
6. 在内层循环结束后，我们通过交换操作将最小值与当前起始位置的元素交换位置。
7. 最后，当外层循环结束后，整个列表就会按照从小到大的顺序进行排列。
8. 代码最后通过调用 `xuanze()` 函数，将排序后的结果赋值给变量 `result` ，并使用 `print` 语句输出排序结果。
9. 运行以上代码，会输出 `[1, 2, 4, 5, 6, 56, 89]` ，表示列表 `data` 已按照从小到大的顺序进行了排序。

【练习】字典实现

项目简介

字典实现

知识模块

- Python 编程语言

知识点

- 字符串处理
- 字典
- 循环语句 for-in
- 分支语句 if-else

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实现一个简单的字典。

解题思路

1. 文本数据初始化：字符串 txt 包含了英文单词和对应的中文翻译。
2. 字符串分割：将字符串 txt 根据换行符和空格进行分割，得到一个包含多个字符串的列表。
3. 构建字典：遍历字符串列表中的每个字符串，去掉可能的 # 标志，然后将单词和翻译分别提取出来，构建一个字典。字典的键是英文单词，值是对应的中文翻译。
4. 用户输入和翻译查找：用户被提示输入一个英文单词。程序检查字典中是否存在该单词，如果存在则输出对应的中文翻译，否则提示单词不存在。

完整代码

```
def func3():
    txt = '''#a
    Trans:art. 一;字母A
    #a.m.
    Trans:n. 上午
    #a/c
    Trans:n. 往来帐户@往来:come - and - go; contact; intercourse@n. 往来帐户
    #aardvark
    Trans:n. 土猪
    #aardwolf
    Trans:n. 土狼
    #aasvogel
    Trans:n. 秃鹰之一种
    #abaci
    Trans:n. 算盘
    #aback
    Trans:ad. 向后地;朝后地
    #abacus
    Trans:n. 算盘
    #abaft
    Trans:ad. 向船尾@prep. 在...后
    #abalone
    Trans:n. 鲍鱼
    #abandon
    Trans:vt. 放弃;沉溺@n. 放任
    #abandoned
    Trans:a. 被抛弃的;自弃的;自甘堕落的
    #abandonee
    Trans:n. 被遗弃者;被委付者
    #abandoner
    Trans:n. 遗弃者;委付者
    #abandonment
    Trans:n. 放弃;自暴自弃;放纵
    #abas
    Trans:vt. 打倒
    #abase
    Trans:vt. 降低...的地位;降低...的品格;贬抑
```

```
#abasement
Trans:n. 贬抑;屈辱;谦卑
#abash
Trans:vt. 使...羞愧;使困窘
#abashment
Trans:n. 羞愧;
#abate
Trans:vt. 缓和;减弱;减少;废除@vi. 缓和;减弱;减少
#abatement
Trans:n. 减少;减轻;缓和
#abatis
Trans:n. 鹿柴;拒木;铁丝网
#abatment
Trans:n. 消除, 减除
#abb
Trans:n. 横丝;纬;羊毛
#abbacy
Trans:n. 大修道院院长之职位;管区;任期
#abbatial
Trans:a. 大修道院的;大修道院长的;女大修道院长的
#abbe
Trans:n. 大修道院院长;僧侣;神父
#abbess
Trans:n. 女修道院院长;女庵主持
#abbey
Trans:n. 修道院
#abbot
Trans:n. 修道院院长;方丈;住持
#abbreviate
Trans:vt. 缩写;使...简略;缩短
#abbreviation
Trans:n. 缩写
#abbreviator
Trans:n. 缩写者
#abc
Trans:n. 基础知识;美国广播公司;澳大利亚广播公司
#abcoulomb
Trans:n. 绝对库伦
#abdicate
Trans:vt. 放弃@vi. 逊位
#abdication
Trans:n. 逊位;弃权;辞职
#abdicator
Trans:n. 放弃者;让位者
#abdomen
Trans:n. 腹部
#abdominal
Trans:a. 腹部的;腹式呼吸;开腹手术
#abduct
Trans:vt. 诱拐;绑走
#abduction
Trans:n. 诱拐
#abductor
Trans:n. 诱拐者
#abe
Trans:n. 亚伯;Abraham 的昵称
#abeam
Trans:ad. 与船的龙骨成直角
#abecedarian
Trans:n. 初学者@a. 字母的;初步的
#abed
Trans:ad. 在床上
#abelmosk
Trans:n. 秋葵
#aberrance
Trans:n. 脱离正道;越轨;脱离常轨
#aberrant
Trans:a. 脱离正道的;脱离常轨的;变体的
#aberration
Trans:n. 越轨;光行差;心理失常;色差
#abestrine
Trans:adj. 石棉的
#abet
Trans:vt. 教唆;帮助'''

data = txt.split("\n    #")
myDict = {}
for item in data:
    # 处理第一个单词字符串
    if item.startswith("#"):
        item = item[1:]
    # 拆分单词项
    k, w = item.split("\n    Trans:")
    myDict[k] = w

inw = input("请输入一个单词：")
if  inw in myDict:
    trans = myDict[inw]
    print(trans)
else:
    print("输入的单词不存在")
```

代码讲解

1. 定义一个函数 `func3`，用于执行单词翻译操作。
2. 准备需要翻译的字符串数据 `txt`，其中包含了单词及其翻译，每个单词和翻译之间使用换行符和 `Trans:` 分隔。
3. 使用 `split` 方法按照换行符和 `#` 分割字符串 `txt`，得到一个包含所有单词及其翻译的列表 `data`。
4. 创建一个空字典 `myDict`，用于存储单词和翻译的键值对。
5. 遍历列表 `data`，对于每个单词项，判断是否以 `#` 开头，如果是，则去掉开头的 `#`；然后使用 `split` 方法按照 `Trans:` 分割为单词和翻译，将其作 为键值对添加到字典 `myDict` 中（单词作为键，翻译作为值）。
6. 提示用户输入一个需要翻译的单词。
7. 如果用户输入的单词存在于字典 `myDict` 中，通过键来获取对应的翻译，并打印出来。
8. 如果用户输入的单词不在字典 `myDict` 中，则打印出"输入的单词不存在"的提示信息。

【练习】字符统计

项目简介

字符统计

知识模块

- Python 编程语言

知识点

- 分支语句
- 循环语句
- 字典

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

从键盘输入一个字符序列，统计，大写字母，小写字母，数字，其它字符各出现次数，将次数保存到字典中。

解题思路

1. 首先，定义了一个名为 `func2()` 的函数，用于统计输入字符序列中的大写字母、小写字母、数字和其他字符出现的次数，并将次数保存到字典中。
2. 在函数内部，使用 `input()` 函数提示用户输入数据，并将输入的字符序列保存在变量 `data` 中。
3. 创建一个字典 `count` ，初始时大写字母、小写字母、数字和其他字符的计数器都设置为0。
4. 将输入的字符序列转换为列表，方便遍历每个字符。
5. 使用 `for` 循环遍历字符列表中的每个字符。
6. 对于每个字符 `c` ，通过一系列的条件判断来确定其属于哪一类字符：
7. 循环结束后，字典 `count` 中保存了大写字母、小写字母、数字和其他字符出现的次数。
8. 使用 `print()` 函数将字典 `count` 打印输出，显示各类字符出现的次数。

完整代码

```
def func2():
    data = input("请输入数据：")
    count = {
        "DX":0,
        "XX":0,
        "SZ":0,
        "QT":0
    }

    data  = list(data)
    print(data)

    for c in data:
        if c >= '0' and c <= '9':
            count["SZ"] += 1
        elif c >= 'A' and c <= 'Z':
            count["DX"] += 1
        elif c >= 'a' and c <= 'z':
            count["XX"] += 1
        else:
            count["QT"] += 1

    print(count)
```

代码讲解

1. 首先，定义了一个名为 `func2()` 的函数。
2. 在函数内部，使用 `input()` 函数提示用户输入数据，并将输入的字符序列保存在变量 `data` 中。
3. 创建了一个字典 `count` ，其中键分别为"DX"、"XX"、"SZ"和"QT"，值均初始化为0。这些键分别代表大写字母、小写字母、数字和其他字符。
4. 将输入的字符序列转换为列表，方便遍历每个字符。
5. 使用 `for` 循环遍历字符列表中的每个字符。
6. 对于每个字符 `c` ，通过一系列的条件判断来确定其属于哪一类字符：

   如果字符是数字，即字符在ASCII码中的范围为'0'到'9'之间（可以直接通过判断 `c` 是否大于等于'0'并且小于等于'9'来判断），则将计数器 `count["SZ"]` 加1，表示数字出现了一次。

   如果字符是大写字母，即字符在ASCII码中的范围为'A'到'Z'之间（可以直接通过判断 `c` 是否大于等于'A'并且小于等于'Z'来判断），则将计数器 `count["DX"]` 加1，表示大写字母出现了一次。

   如果字符是小写字母，即字符在ASCII码中的范围为'a'到'z'之间（可以直接通过判断 `c` 是否大于等于'a'并且小于等于'z'来判断），则将计数器 `count["XX"]` 加1，表示小写字母出现了一次。

   否则，即为其他字符，将计数器 `count["QT"]` 加1，表示其他字符出现了一次。
7. 循环结束后，字典 `count` 中保存了大写字母、小写字母、数字和其他字符出现的次数。
8. 使用 `print()` 函数将字典 `count` 打印输出，显示各类字符出现的次数。

【练习】整数转二进制数

项目简介

整数转二进制数

知识模块

- Python 编程语言

知识点

- 循环语句
- 运算符

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，输入一个整数，计算该整数的二进制表示形式

解题思路

1. 使用 `input()` 函数获取用户输入的内容，并将其转换为整数类型。
2. 创建变量，用于存储最终的二进制表示形式。
3. 使用循环来计算整数的二进制表示。
4. 输出最终的二进制表示形式，可以使用 `print()` 函数将结果打印出来。

完整代码

```
def func5():
    result = ""
    data = int(input("Num:"))
    while data != 0:
        tmp = data % 2
        data = data // 2
        result = str(tmp) + result
    print(result)
```

代码讲解

1. 首先，我们初始化一个空字符串 `result`，用于存储最终的二进制表示形式。
2. 然后，使用 `input` 函数获取用户输入的整数，并通过 `int` 函数将其转换为整数类型，赋值给变量 `data`。
3. 接下来，使用 `while` 循环进行二进制转换。在每次循环中，我们通过 `data % 2` 取余操作得到当前位置的二进制值，并将其存储在临时变量 `tmp` 中。然后，使用 `data // 2` 进行整除操作得到下一个位置的二进制值。最后，通过 `result = str(tmp) + result` 将当前位置的二进制值转换为字符串，并添加到结果字符串 `result` 的前面。
4. 当整数 `data` 为 0 时，循环终止，表示二进制转换完成。最后，我们使用 `print(result)` 将最终的二进制表示形式打印输出到控制台。

循环嵌套

循环嵌套

在很多实际场景中，单层的循环并不能解决问题，比如二维列表数据的遍历，行列数据的展示等，此时就需要在一个循环的循环体中嵌入另外一个循环处理问题。 此时就形式了嵌套循环，甚至可能还会产生三层甚至更多的嵌套形式。

循环嵌套特征

- 循环嵌套不局限于某种循环形式，可任意相互嵌套
- 外层循环循环一次，内层循环循环一轮

示例1：使用 `for-in` 循环遍历二维列表

```
data = [
    [1,2,3,4,5,6,7,8,9],
    ["A","B","C","D","E"],
    ["Hello","World","Python","Hogwarts"]
]
for item in data:
    for el in item:
        print(el)
```

示例2：使用 `while` 循环遍历二维列表

```
data = [
    [1,2,3,4,5,6,7,8,9],
    ["A","B","C","D","E"],
    ["Hello","World","Python","Hogwarts"]
]
l1 = len(data)
i = 0
while i < l1:
    item = data[i]
    l2 = len(item)
    j = 0
    while(j < l2):
        el = item[j]
        print(el)
        j += 1
    i += 1
```

示例3: 输出九九乘法表的结果

```
for i in range(1,10):
    for j in range(1,i+1):
        print(i*j, end=" ")
    print()
```

实战作业

1. 使用 `for-in` 内嵌 `while` 访问示例中的二维列表
2. 使用 `while` 内嵌 `for-in` 访问示例中的二维列表
3. 实现 九九乘法表标准格式输出

```
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
```

【练习】数字组合

项目简介

数字组合

知识模块

- Python 编程语言

知识点

- 分支语句-if
- 循环语句-for-in
- 循环嵌套

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个 Python 程序，输出所有由数字1、2、3、4 组成的互不相同且无重复数字的三位数。即个位，十位，百位互不相同且无重复数字。

解题思路

1. 使用三重嵌套循环：用来生成所有可能的三位数的组合。每个循环变量代表一个位上的数字。
2. 条件判断：用来排除数字相同的情况，确保三位数是互不相同且无重复数字的。
3. 输出符合条件的数字

完整代码

```
for i in range(1, 5):
    for j in range(1, 5):
        for k in range(1, 5):
            # 条件判断排除重复数字的排列
            if (i != k) and (i != j) and (j != k):
                print(i, j, k)
```

代码讲解

1. 三复嵌套循环：i 表示百位数字，j 表示十位数字，k 表示个位数字，生成所有可能的排列组合。
2. 条件判断：在内层循环中，使用条件判断来排除出现重复数字的排列。通过 `(i != k) and (i != j) and (j != k)` 条件，确保了 i、j、k 三个数字都不相等，从而生成了互不相同的三位数。
3. 输出组合：在满足条件的情况下，通过 `print(i, j, k)` 将当前的三位数排列输出。

循环跳转

循环跳转

在使用循环进行实现逻辑时，并不是每次都会将循环执行完毕，或者循环根本没有结束的条件，此时，在循环过程中，期望达到某个条件时，结束循环继续向后执行，此时，就可以使用循环跳转来完成此功能。

Python中提供两种循环跳转语句：

- `break` 结束当前循环
- `continue` 结束本次循环

在讲解之前，先来了解一种特殊的循环方式。

死循环

死循环是指循环条件恒等为真，循环会一直执行下去，死循环并不是一种错误语法，反之，有效使用死循环可以解决很多问题。但实际使用中，还是要尽量避免死循环的使用。

```
while True:
    print("Hogwarts!")
```

一般，在不确定循环次数的情况下，可以使用死循环来实现。比如web服务器监听处理客户端的请求连接，服务器并不知道会有多少客户端会连接，此时就可以使用死循环实现。

示例： 从键盘持续输入数据，每输入一次，将数据添加到列表中保存

```
data = []
while True:
    d = input("请输入数据：")
    data.append(d)
    print(data)
```

此程序只为演示死循环的使用方式，并无实际意义。

break 语句

break 语句可以中止当前的while 或for-in循环，无论循环还有多少次没有循环，都会无条件结束循环后，跳转到循环后的语句继续执行。

以生活中的跑步为例，计划跑 5 圈，可是在跑到第 2 圈的时候，接到了室友的电话叫去吃大餐，于是果断停下来，中止跑步，这就相当于使用 break 语句提前中止了循环。

break 语句的语法比较简单，只需要在相应的 while 或 for 语句中加入即可。

前面的示例自用 break 可以优化为,当输入 `bye` 时，结束输入

```
data = []
while True:
    d = input("请输入数据：")
    if d == "bye":
        break
    data.append(d)
    print(data)
```

需要注意的是，break 语句出现在多层循环中时，所处在哪个循环的循环体内，就跳出哪个循环，更外层的循环不受影响。

```
for i in range(2):
    for j in range(5):
        print("Hello", j)
        if j == 2:
            break
```

continue 语句

continue 语句的作用用来中止本次循环而提前进入下一次循环中，循环体中未执行完的代码不在执行。

仍然以操场跑步为例，计划跑 5 圈，但是在跑完每圈的前一百米就回到起点重新跑，后面的不太跑了。

```
for i in range(10):
    print("*" * 10)
```

```
    print("i=",i)
    if i % 3 == 0:
        continue
    print("i*10= ", i*10)
```

大多数情况下，可以通过改变条件的方式，避免continue的使用。而代码更简洁。

```
for i in range(10):
    print("*" * 10)
    print("i=",i)
    if i % 3 != 0:
        print("i*10= ", i*10)
```

loop-else

在Python中，不只if 语句可以使用else, 循环语句也可以使用else。

当一个循环没有被break中断而正常循环结束，就会执行else后的代码块。

```
for i in range(5):
    print(i)
else:
    print("Over")
```

如果循环被断，则不会执行else后的语句

```
for i in range(5):
    print(i)
    if i == 2:
        break
else:
    print("Over")
```

在某些场景下，else 可以简化代码。

```
names = ["tom","jack","rose","tony"]

inName = input("请输入要查找的姓名：")

# 不使用Else形式
flag = False
for name in names:
    if name == inName:
        print("找到了")
        flag = True
        break
if not flag:
    print("没找到")

# 使用Else形式
for name in names:
    if name == inName:
        print("找到了")
        break
else:
    print("没找到")
```

实战作业

推导式

推导式

Python 推导式是一种独特的数据处理方式，可以从一个数据序列构建另一个新的数据序列的结构体。

Python 支持各种数据结构的推导式：

- 元组(tuple)推导式
- 列表(list)推导式
- 字典(dict)推导式
- 集合(set)推导式

元组推导式

元组推导式可以利用 range 区间、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的元组。

元组推导式基本格式：

```
(out_exp_res for item in Sequence )
```

或

```
(out_exp_res for item in Sequence if conditional )
```

- `out_exp_res`：生成元素表达式，可以是有返回值的函数。
- `for out_exp in Sequence`：迭代 Sequence 将 out_exp 传入到 out_exp_res 表达式中。
- `if condition`：条件语句，可以过滤Sequence中不符合条件的值。

```
# 简单的元组推导式
t1 = (x for x in range(1,10))
# 生成128位ASCII码元组
t2 = (chr(x) for x in range(128))
# 生成100以内能被7整除所有数字的元组
t3 = (x for x in range(100) if x%7==0)
# 生成99乘法表结果元组
t4 = (x*y for x in range(1,10) for y in range(1, x+1))
words = ["apple", "banana", "cherry"]
upper_words = (word.upper() for word in words)
```

列表推导式

列表推导式与元组推导式的区别：

- 格式上外部由圆括号包裹的表达式改为中括号

列表推导式格式为：

```
[out_exp_res for item in Sequence ]
```

或

```
[out_exp_res for item in Sequence if conditional ]
```

- `out_exp_res`：生成元素表达式，可以是有返回值的函数。
- `for out_exp in Sequence`：迭代 Sequence 将 out_exp 传入到 out_exp_res 表达式中。
- `if condition`：条件语句，可以过滤Sequence中不符合条件的值。

```
# 简单的元组推导式
l1 = [x for x in range(1,10)]
# 生成128位ASCII码元组
l2 = [chr(x) for x in range(128)]
# 生成100以内能被7整除所有数字的元组
l3 = [x for x in range(100) if x%7==0]
# 生成99乘法表结果元组
l4 = [x*y for x in range(1,10) for y in range(1, x+1)]
# 将列表中的字符串转换为大写
```

```
words = ["apple", "banana", "cherry"]
upper_words = [word.upper() for word in words]
```

字典推导式

字典推导式与前两种推导式的区别

- 推导式使用花括号包裹
- 结果表达式需要使用 key-value 形式

字典推导基本格式：

```
{ key_expr: value_expr for value in collection }
```

或

```
{ key_expr: value_expr for value in collection if condition }
```

```
names = ['Bob','Tom','alice','Jerry','Wendy','Smith']
# 将列表中各字符串值为键，各字符串的长度为值，组成键值对
newdict = {name:len(name) for name in names}
```

集合推导式

集合推导式与字典推导式的区别在于结果表达式是单一结果，不是key-value形式。

集合推导式基本格式：

```
{ expression for item in Sequence }
```
或
```
{ expression for item in Sequence if conditional }
```

```
data = ['Bob','123','Tom','ab123','alice','123abc','Jerry','456','Wendy','554','Smith']
# 筛选列表中的数字字符串
newset = {n for n in data if n.isdigit()}
```

推导式的优点

- 简洁高效：推导式允许在一行代码中完成复杂的生成操作，避免了使用显式的循环和临时变量的繁琐过程。这样可以大大减少代码量，并提高编码效率。
- 可读性好：推导式使用简洁的语法，将迭代、条件判断和表达式结合在一起，使得代码更加紧凑和易于理解。它提供了一种清晰、直观的方式来表达列表生成的逻辑，使得代码更加可读性强，降低了出错的可能性。
- 灵活性强：推导式非常灵活，可以根据需要添加条件判断、多个迭代变量和嵌套结构，以满足不同的需求。这使得推导式适用于各种场景，从简单的数据转换到复杂的筛选和操作，都可以通过简单而直观的语法实现。

总的来说，推导式提供了一种简洁高效、可读性好和灵活性强的方法来处理列表数据。它是Python中强大而常用的特性之一，为开发者提供了更好的编码体验和效率。

1.1.7 Python 函数

函数返回值与参数处理

函数返回值与参数处理

在前面讲解了函数的基本定义和使用。通过一段时间的使用，对函数的使用有了一定的了解。

但是实际函数使用时，基本的定义和调用还远远没有达到函数的使用要求。

函数的主要作用是实现模块化代码，减少代码在程序中的冗余，提高代码的复用率。

函数通过接收一定的数据，根据业务逻辑进行相应的处理，然后将处理结果返回给调用者。

实现函数要基于高内聚，低耦合的思想。

高内聚低耦合指的是函数内部的代码要有高内聚性，而函数之间要保持低耦合性。

高内聚性指的是函数内部的代码逻辑和功能彼此相关，实现的是一个具体的功能或目标，各个部分之间密切配合。

一个高内聚的函数通常只完成一个明确的任务，并且其内部的代码逻辑清晰、简洁。

低耦合性指的是函数之间的依赖关系较弱，相对独立，一个函数的改动不会对其他函数造成过多的影响。

低耦合的函数在设计上应该尽量减少使用全局变量或直接修改其他函数的参数，而是通过参数传递和返回值来进行数据交互。

通过遵循函数的高内聚低耦合原则，可以带来以下优点：

1. 可维护性：函数内部的代码高度相关和一致，使得函数更易于理解、调试和修改，减少了不相关的代码耦合。
2. 可测试性：函数的高内聚性使得它们更易于独立地进行单元测试，因为它们具有清晰的功能和输入输出。
3. 可复用性：函数之间的低耦合性使得它们可以在不同的上下文中被重复使用，提高了代码的复用性。
4. 灵活性：函数之间相互独立且功能明确，使得系统更容易进行扩展和修改，不会对其他函数产生很大的影响。

在设计函数时，应该关注其功能的内聚性，确保函数只完成一个特定的任务，并且将其功能按照模块化的方式进行拆分。同时，尽量减少函数之间的直接依赖和共享状态，通过参数传递和返回值来进行数据交互，以减少函数之间的耦合度。

通过保持函数的高内聚低耦合，能够提高代码的可维护性、可测试性和可复用性，同时使得系统更具灵活性和可扩展性。

函数返回值

在设计一个函数时，可以通过函数的返回值，将函数功能的处理结果，返回给调用者。

需要注意的是，之前函数的使用单纯只是为了帮助理解函数的定义和使用过程，所以函数中没有体现任何返回值。

而一个真正完整的函数，需要返回值将结果返回给调用者，而不是直接通过 print函数将其输出。

比如：存在一个可以查找列表中最大数值的函数，而刚好调用者有需求在一个列表中要找到最大值，并且进行后续操作。如果此时查找最大值函数没有返回结果而是直接输出的话，对于调用者来说，该函数没有任何意义。

Python 中使用 return 返回函数的结果，同时，return 也具有结束函数的作用。

return 结束函数执行

```
def show():
    print("循环前输出内容")
    for i in range(10):
        print(i)
        if i == 2:
            return
    print("循环后输出内容")

print("函数调用前输出内容")
show()
print("函数调用后输出内容")
```

上面的代码中，在循环变量 i 值为 2 时，执行了 return ,此时，函数会立即结束，返回到函数调用语句处继续向后继续执行。

函数只能返回一个值

一个函数只能返回一个结果值，当需要返回多个值时，需要进行包装处理。

```
def getString():
    s = input("请输入一个字符串:")
    return s

result = getString()
print(result.upper())
```

此函数用来从键盘获取一个字符串，并返回给了调用者。调用者得到结果后进行了大写处理。

如果有两个或更多的值想返回，都写在 return 后行不行呢？

```
def getTwoNum():
    a = int(input("请输入第一个数字："))
    b = int(input("请输入第二个数字："))
    return a, b

m, n = getTwoNum()
print(m, n)
```

通过代码发现，函数执行结果和前面介绍的理论并不符合，但实际上，该函数返回的依然是一个值，只是Python在执行时，自动做了组包和解包操作。

```
def getTwoNum():
    a = int(input("请输入第一个数字："))
    b = int(input("请输入第二个数字："))
    return a, b

result = getTwoNum()
print(result)
print(type(result))
```

通过代码执行结果发现，使用一个变量，也可以接收返回的两个数据，并且变成了一个元组。这就是组包操作。

组包与解包

```
# 组包
nums = 1,2,3,4,5
print(nums)
print(type(nums))

# 解包
a,b,c,d,e = nums
print(a,b,c,d,e)
```

通过代码可以发现，当将多个值，同时赋给一个变量时，Python会进行自动组包操作，将所有的数字组合成一个元组，再将元组赋值给变量。当使用一个元组为多个变量进行赋值时，Python会将元组中的元素值，依次赋值给变量，这称为解包操作。

由此可以看出，上一小节中的函数依然返回的是一个结果值，结果值的类型是一个元组。

多个return语句

在一个函数中，允许有多个return语句，可以实现在不同的条件下，返回不同的值，但同一时刻，只能有一个return语句被执行。

```
def multiReturn():
    return 1
    return 2
    return 3
    return 4
    return 5

print(multiReturn())
print(multiReturn())
print(multiReturn())
print(multiReturn())
print(multiReturn())
```

通过代码发现，虽然函数提供了5条return语句，并返回了不同的结果值，在调用时，也调用了五次，但实际，5次的调用结果都是相同的，都为1。

参数传递

参数是指函数在完成特定功能时，需要外部传递的数据。

比如，现实生活中去餐厅吃饭点的菜单，打车时告诉司机的目的地，就医时告诉医生的症状等这些都是参数。

回到程序中，前面用过很多次的 len() 方法，这个方法用来对指定的数据返回其长度或元素个数，那么这个函数的功能是用来获取长度，测谁的长度，该函数并不关心，只要调用者指定的数据可测，函数就返回一个长度，而函数并不关心，被测试的数据是什么。

再回到打车的例子，出租车只负责将人从一个地方拉到另一个地方，司机只关心出发地和目的地信息，而具体是谁去坐车，坐车人什么身份，为什么在出发地，去目的地做什么，这些司机都不关系，司机只负责将人拉到目的地，任务就完成了。

在程序中，函数调用时指定数据的过程，称为参数传递，这时有四个概念，如下：

- 主调函数：主动调用其它函数执行的称为主调函数。
- 被调函数：被动调用执行的函数称为被调函数。
- 实际参数：在调用函数的时候，函数名称后面括号中的数据即为实际参数，简称实参，通俗讲就是实际值。
- 形式参数：在定义函数的时候，函数名称后面括号中的变量即为形式参数，称称形参，通俗讲就是一个记号。

在调用函数时，实参数据会依次传递给形参。

```
# name, age, gender 为形参
def info(name, age, gender):
    print(f"我叫{name}, 年龄{age}岁, 性别{gender}")

# 调用时的数据为实参
info("Tom", 22, "男")
info("Rose", 23, "女")
```

位置参数

位置参数也称为必备参数，必须按照正确的顺序传到函数中，即调用时的数量和位置必须 和定义时是一样的。

- 数量必须与定义时一致，在调用函数时，指定的实际参数的数量必须与形式参数的数量一致，否则将抛出 TypeError 异常，提示缺少必要的位置参数（missing x required positional argument）。
- 位置必须与定义时一致，在调用函数时，指定的实际参数的位置必须与形式参数的位置一致，否则将抛出 TypeError 异常或者结果与预期不符的情况，例如入参的位置 1 需要一个 int 类型参数，而入参 2 位置需要一个 str 类型的参数，如果传递的位置不正确，那么 str 类型数据传递进去之后会报类型错误的异常；位置参数是会按照顺序进行参数的替换的。

```
def printMsg(n, msg):
    for i in range(n):
        print(f'第{i+1}次输出{msg}')

# 正确使用位置参数
printMsg(5, "Hogworts")
# 错误使用位置参数
printMsg("Hogworts", 5)
```

关键字参数

关键字参数是指使用形式参数的名字来确定输入的参数值，通过该方式指定实际参数时，不再需要与形式参数的位置完全一致，只要将参数名写正确即可，这样可以避免用户需要牢记参数位置的麻烦，使得函数的调用和参数传递更加灵活方便。

以前面定义的函数传参为例：

```
def printMsg(n, msg):
    for i in range(n):
        print(f'第{i+1}次输出{msg}')

# 关键字参数
printMsg(n=5, msg="Hogworts")
printMsg(msg="Hogworts", n=5)
```

默认值参数

在定义函数时，形参可以定义变量一样进行赋值，这个值就是默认该参数的默认值，调用函数时，如果指定了该参数的数据，则使用指定的数据，如果没有指定该参数的数据，则使用默认值的数据。

```
# 实现一个乘方函数
def my_power(m, n=2):
    return m ** n

# 使用指定的参数
print(my_power(2, 3))
# 使用默认值参数
print(my_power(2))
```

需要注意的是，指定默认值的形式参数必须放在所有未指定默认值参数的后面，否则会产生语法错误

```
def show(a, b=2, c):
    print(a, b, c)
```

可变参数

在程序运行过程中，所定义的函数，可能不能确定需要接收多少参数，有可能很多，有可能很少甚至没有，此时，就不能通过定义固定个数的参数来接收数据，这会产生两个问题，如果实参比形参多，则没有足够的形参支接收保存实参数据，如果实参比形参少，则形参接收不到数据无法进行函数的数据处理。

Python中提供了可变参数的概念用来解决参数个数不确定的的情况。

可变位置参数

- Python使用 `*args` 参数做为形参，接收不确定个数的位置参数。
- `*args` 将接收到的任意多个实际参数放到一个元组中。

```
# 不确定个数数字求和
def my_sum(*args):
    print(args)
    print(*args)
    print(type(args))
    s = 0
    for i in args:
        s += i

    print(s)
    print("*" * 10)

my_sum(1,2,3)
my_sum(1,2,3,4)
my_sum(1,2,3,4,5)
my_sum(1,2,3,4,5,6)
```

可变关键字参数

- Python使用 `**kwargs` 参数做为形参，接收不确定个数的关键字参数。
- `**kwargs` 将接收到的任意多个实际参数放到一个字典中。

```
# 定义可变关键字参数
def print_info(**kwargs):
    print(kwargs)
    print(type(kwargs))
    for k,v in kwargs.items():
        print(k, v)
    print("*" * 10)

print_info(Tom=18, Jim=20, Lily=12)
print_info(name="tom",age=22,gender="male",address="BeiJing")
```

混合参数

当定义函数时，参数列表中出现了多种类型的参数，定义时需要注意参数的定义顺序，如果顺序使用不正确，在调用函数时，可能会报错。

正确顺序的定义为：`位置参数`, `可变位置参数`, `默认值参数`, `可变关键字参数`

```
def info(name1, name2, *args, age1=18, age2=21, **kwargs):
    print("Positional Arguments:")
    print("name1:", name1)
    print("name2:", name2)
    print("args:", args)
    print("Keyword Arguments:")
    print("age1:", age1)
    print("age2:", age2)
    print("kwargs:", kwargs)

info("Alice", "Bob", "Charlie", "Dave", age2=30, occupation="Engineer", city="New York")
info("Alice", "Bob", "Charlie", "Dave", age1=25, age2=30, occupation="Engineer", city="New York")
```

一般函数在不确定参数的情况下，会将上面的形式简化定义，用来接收任意数量的参数。

```
def funcname(*args, **kwargs):
    pass
```

在此定义形式中，使用 `*args` 接收所有的位置参数，使用 `**kwargs` 接收所有的关键字参数。

需要注意的是，传递参数时，需要先传递完所有的位置参数后，才能传递关键字参数。

变量作用域

变量作用域

作用域就是一个 Python 程序可以直接访问命名空间的正文区域。

在一个 Python 程序中，直接访问一个变量，会从内到外依次访问所有的作用域直到找到，否则会报未定义的错误。

Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。Python 的作用域一共有4种，分别是：

有四种作用域：

- `L (Local)`：最内层，包含局部变量，比如一个函数/方法内部。
- `E (Enclosing)`：包含了非局部(non-local)也非全局(non-global)的变量，一般在闭包中出现。
- `G (Global)`：当前脚本的最外层，比如当前模块的全局变量。
- `B (Built-in)`： 包含了内建的变量/关键字等，最后被搜索。

查找规则顺序：`L -> E -> G -> B`。

在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内置中找。

根据作用域的不同，变量可分为局部变量和全局全量。

局部变量

局部变量是指在函数内部定义并使用的变量，它只在函数内部有效，即函数内部的名字只在函数运行时才会创建，在函数运行之前或者运行完毕之后，所有的名字就都不存在了。 所以，如果在函数外部使用函数内部定义的变量，就会抛出 NameError 异常。

```
def show():
    # 局部变量
    a = 10
    print(a)

# 在函数外无法访问局部变量a
# print(a)
# 只能通过调用函数使用局部变量
show()
# 函数外无法访问局部变量a
# print(a)
```

定义在函数内部的局部变量，只在函数内部可以被访问。局部变量在函数被调用时被创建，在函数执行结束后被销毁。

全局变量

定义在函数外之外的变量为全局变量，全局变量可以在当前程序文件的任何位置进行访问 。

```
# 全局变量定义
m = 20
def show1():
    # 使用全局变量
    print("show1:", m)

def show2():
    # 使用全局变量
    print("show2:", m)

# 使用全局变量
print(m)
show1()
show2()
# 使用全局变量
print(m)
```

当全局变量和局部变量同名时，根据作用域的查找规则顺序，内部变量会屏蔽全局变量。

```
# 全局变量定义
m = 10
def show():
    # 局部变量与全局变量同名
    m = "ABC"
    # 使用局部变量
    print("show:", m)
```

```
# 使用全局变量
print(m)
show()
# 使用全局变量
print(m)
```

从上面的代码中可以看出，如果想在函数内部修改全局变量时，会被解释成同名变量屏蔽，此时，需要使用 `global` 关键字

```
# 全局变量定义
m = 10
def show():
    # 声明此变量是函数外定义的全局变量
    global m
    # 修改全局变量的值
    m = "ABC"
    # 使用全局变量
    print("show:", m)

# 使用全局变量
print(m)
show()
# 使用全局变量
print(m)
```

与局部变量对应，全局变量为能够作用于函数内外的变量。全局变量主要有两种情况。

情况一：如果一个变量在函数体外定义，那么不仅在函数外可以访问到，在函数内也可以访问到。在函数体以外定义的变量是全局变量。

改造一下前面局部变量的例子

```
msg = "学习测试开发来霍格沃兹测试学社" # 创建一个局部变量并赋值
def var_demo():
    print("函数体内全局变量 msg 的值为", msg) # 输出局部变量 msg 的值

var_demo()    # 调用函数
print("函数体外全局变量 msg 的值为", msg) # 在函数体外输出局部变量的值
```

上面的例子中，在 var_demo 函数中创建之前就定义了一个 msg 变量，在函数之外属于全局变量；在函数的内部直接使用该变量是正常的，能正常输出赋予的值；在函数体之外进行变量值的输出的时候，也会正常的输出，不会抛出 NameError 异常。注意，当局部变量与全局变量重名时，对函数体内的变量进行赋值后，操作的是局部变量，不影响函数体外的变量。

情况二：如果一个变量在函数体内定义，并且使用 global 关键字修饰后，该变量也就变为全局变量。在函数体外可以访问该变量，并且在函数体内还可以对其进行修改。

改造一下全局变量的例子

```
msg = "学习测试开发来霍格沃兹测试学社" # 创建一个局部变量并赋值
print("函数体外全局变量 msg 的值为", msg) # 在函数体外输出局部变量的值
def var_demo():
    msg = "霍格沃兹测试学社值得信赖" # 创建一个局部变量并赋值
    print("函数体内变量 msg 的值为", msg) # 输出局部变量 msg 的值

var_demo()    # 调用函数
print("函数体外全局变量 msg 的值为", msg) # 在函数体外输出局部变量的值
```

改造成为当前的模式之后实际上局部变量与全局变量重名时，对函数体内的变量进行赋值后，操作的是局部变量，不影响函数体外的变量。

如果想要实现函数内容修改全局变量的值就需要借助 global 来实现

进一步改动全局变量的例子

```
msg = "学习测试开发来霍格沃兹测试学社" # 创建一个局部变量并赋值
print("函数体外全局变量 msg 的值为", msg) # 在函数体外输出局部变量的值
def var_demo():
    global msg
    msg = "霍格沃兹测试学社值得信赖" # 此时已经申明了 msg 为全局变量
    print("函数体内变量 msg 的值为", msg) # 输出局部变量 msg 的值

var_demo()    # 调用函数
print("函数体外全局变量 msg 的值为", msg) # 在函数体外输出局部变量的值
```

全局变量和局部变量的优缺点

全局变量的优点：

1. 全局变量在整个程序中都可访问，方便在不同的函数或模块之间共享数据。
2. 全局变量可以保存程序运行期间的状态，比如计数器或状态标志。

全局变量的缺点：

1. 全局变量的作用范围很大，容易被误修改，导致程序出现错误。
2. 在多个函数中使用同名的全局变量时，很容易出现混淆和命名冲突的问题。
3. 全局变量的使用不宜过多，因为它们会占用内存资源。如果程序中有太多的全局变量，会增加维护和调试的难度。

局部变量的优点：

1. 局部变量的作用范围限于所在的函数或代码块中，不会对其他函数或模块造成影响。这样可以防止变量的误修改或命名冲突。
2. 局部变量仅在需要时才会被创建和销毁，节省了内存资源。

局部变量的缺点：

1. 局部变量的作用范围有限，不能在其定义范围外直接访问。如果需要在多个函数之间共享数据，就不能使用局部变量，需要采用其他的方式来传递数据。
2. 当函数中定义了较多的局部变量时，可能会使代码显得冗长和难以阅读，增加了编写和维护的复杂性。

在编写程序时，应根据具体的需求来选择使用全局变量还是局部变量。如果需要在多个函数中共享数据，可以选择使用全局变量；如果只在某个函数内部使用的变量，可以选择使用局部变量。同时，需要注意合理命名变量，以避免命名冲突和混淆。

【练习】素数

项目简介

素数

知识模块

- Python 编程语言

知识点

- 运算符
- 循环语句-for-in
- 分支语句-if
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，输入一个正整数，判断这个数是否为素数（质数）。素数是指只能被1和它本身整除的正整数。

解题思路

1. 获取用户输入的正整数。
2. 判断输入的数是否小于等于1，如果是，则不是素数。
3. 对于大于1的数，从2开始到这个数的平方根（包括平方根），逐个检查是否能整除。如果存在能整除的数，则不是素数。
4. 如果没有找到能整除的数，那么这个数是素数。

完整代码

```
import math

# 用户输入正整数
num = int(input("请输入一个正整数："))

# 判断是否为素数
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

# 输出结果
if is_prime(num):
    print(f"{num} 是素数")
else:
    print(f"{num} 不是素数")
```

代码讲解

1. `import math`：导入 Python 的 `math` 模块，用于计算平方根。
2. `num = int(input("请输入一个正整数："))`：获取用户输入的正整数并将其转换为整数类型，存储在变量 `num` 中。
3. `def is_prime(n)`：定义一个名为 `is_prime` 的函数，用于判断是否为素数。
4. `if n <= 1`：如果输入的数小于等于1，返回 `False`，因为1和负数不是素数。
5. `for i in range(2, int(math.sqrt(n)) + 1)`：通过循环从2到数的平方根（包括平方根）遍历每个数，以检查是否存在能整除的因子。
6. `if n % i == 0`：如果存在可以整除的因子，返回 `False`，说明不是素数。
7. `return True`：如果循环结束后都没有找到可以整除的因子，返回 `True`，说明是素数。
8. 最后，根据 `is_prime` 函数的返回值输出相应的信息，判断输入的数是否为素数。

匿名函数

匿名函数

匿名函数是指没有名字的函数，应用在需要一个函数，但是又不想费神去命名这个函数的场合。在通常情况下，这样的函数只使用一次。

在 Python 中使用 lambda 表达式创建匿名函数。

- Lambda 函数可用于任何需要函数对象的地方。
- 在语法上，匿名函数只能是单个表达式。
- 在语义上，它只是常规函数定义的语法糖。
- lambda 表达式中不能使用 if,for,while, return等语句, 但可以使用三目运算符

lambda 表达式

lambda 表达式的语法格式如下：

```
result = lambda [arg1 [, arg2, .... , argn]]: expression
```

- result：用于保存 lambda 表达式的引用
- [arg1 [, arg2, ... , argn]]：可选，指定要传递的参数列表，多个参数间使用英文的逗号， 进行分隔。
- expression：必选，指定一个实现具体功能的表达式。如果有参数，那么在该表达式中将应用这些参数。

```
def add(n1, n2):
    return n1 + n2

result = add(1,2)
print(result)


add = lambda x,y: x+y
result = add(2,3)
print(result)

func = lambda x: x**2  if x > 3 else x**3
print(func(3))
```

使用场景

- 需要一个函数，但是又不想费神去命名这个函数
- 通常在这个函数只使用一次的场景下
- 可以指定短小的回调函数

比如，在学习列表时的sort()排序方法，如果是简单的基本数据类型的数据，可以直接进行排序，但如果是复杂结构的数据，需要根据自定义的规则进行排序，此时就可以使用lambda。

```
# 待排序的数据
students = [
    {'name': 'Alice', 'id': '1001', 'class': 'Class1'},
    {'name': 'Eve', 'id': '1005', 'class': 'Class2'},
    {'name': 'Charlie', 'id': '1003', 'class': 'Class1'},
    {'name': 'David', 'id': '1004', 'class': 'Class2'},
    {'name': 'Bob', 'id': '1002', 'class': 'Class1'},
    {'name': 'Frank', 'id': '1006', 'class': 'Class2'}
]
# TypeError: '<' not supported between instances of 'dict' and 'dict'
# students.sort()

# 以名字排序
students.sort(key=lambda stu: stu["name"])
for s in students:
    print(s)

# 以ID降序排序
students.sort(key=lambda stu: stu["id"],reverse=True)
for s in students:
    print(s)
```

Sorted函数实现原理

```
students = [
    {'name': 'Alice', 'id': '1001', 'class': 'Class1'},
```

```
    {'name': 'Eve', 'id': '1005', 'class': 'Class2'},
    {'name': 'Charlie', 'id': '1003', 'class': 'Class1'},
    {'name': 'David', 'id': '1004', 'class': 'Class2'},
    {'name': 'Bob', 'id': '1002', 'class': 'Class1'},
    {'name': 'Frank', 'id': '1006', 'class': 'Class2'}
]

def mySorted(obj, key=None, reverse=False):
    newStus = []
    for s in students:
        for n in newStus:
            if key:
                if(key(s) < key(n)):
                    idx = newStus.index(n)
                    newStus.insert(idx, s)
                    break
            else:
                if (s < n):
                    idx = newStus.index(n)
                    newStus.insert(idx, s)
                    break
        else:
            newStus.append(s)

    return newStus if reverse else newStus[::-1]

# students = mySorted(students, key=lambda s: s["name"])
# students.sort(key=lambda s: s["name"])
students = [1,4,2,6,7,8,4,3,3]
students = mySorted(students, reverse=True)
print(students)
for s in students:
    print(s)
```


递归算法

递归调用

有一个很古老的故事是这么讲的 从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事！故事是什么呢？从前有座山，山里有座庙，庙里有个老和尚正在给小和尚讲故事！故事是什么呢.....

除非讲故事的人自己停下来不讲了，不然这个故事可以无限讲下去，原因就是故事嵌套的故事就是故事本身，实际上这个就是语言的递归一个例子。但上面的例子是没有结束符的，如果设计到代码中就会陷入无限循环执行的死循环状态，无法为程序的设计人员解决事务；当涉及解决问题时，递归是一种非常有用的编程技巧，为它设计一个出口，它允许函数调用自身以解决更小规模的问题，直到达到基本情况为止。

递归的基本原则

递归函数通常遵循以下原则：

- 定义基本情况：确定一个或多个输入的特殊情况，当满足这些条件时，递归函数将直接返回结果而不再调用自身。
- 减小问题规模：通过调用自身来解决一个规模更小的问题，这样每次递归调用都在问题规模上取得了进展。也就是需要一个已定义好的规则来使其它非基本的情况转化为基本情况。
- 终止条件：递归函数必须包含能够导致函数不再递归调用的条件，以避免无限递归。

递归使用举例-阶乘

数学中有一个比较经典的递归算式就是求阶乘

一个正整数的阶乘（factorial）是所有小于及等于该数的正整数的积，并且0的阶乘为1。自然数n的阶乘写作n!。 对阶乘的定义进行分析：

- 首先定义 f(n) 为阶乘函数。它的结果就是阶乘计算之后的值。
- 从定义中能得到基本情况为: f(0) = 1, f(1) = 1
- 其它情况: `f(2) = 2*1 = 2*f(1),f(3) = 3*2*1 = 3*f(2)` 即 `f(n) = f(n-1) * n`

接下来把分析的结果转换成 Python 代码

```
def f(n):
    """
  实现计算 n 的阶乘
    return：n 阶乘计算之后的值
    """
    if n == 0 or n == 1:
        # 对应基本情况
        return 1
    return f(n - 1) * n  # 对应递归情况
```

大多数情况下的递归操作都可以使用循环所替代。

在使用递归时，要注意避免陷入无限调用而产生的内存溢出。

【练习】阶乘

项目简介

阶乘

知识模块

- Python 编程语言

知识点

- 分支语句-if
- 递归算法
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，使用递归算法，计算给定正整数n的阶乘。阶乘的定义是从1到n所有正整数的乘积。例如3的阶乘为1*2*3=6。

解题思路

1. 当 n 等于 0 或 1 时，阶乘的值为 1。因为 0! 和 1! 都等于 1。
2. 对于大于 1 的正整数 n，n! 等于 n 乘以 (n-1)!。我们可以使用递归调用来计算 (n-1)!，然后将结果乘以 n，从而得到 n!。

完整代码

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

n = int(input("请输入要计算阶乘的正整数："))
result = factorial(n)
print(f"{n}的阶乘是：{result}")
```

代码讲解

1. 定义了一个名为 `factorial` 的函数，该函数接受一个参数 `n`，表示要计算阶乘的正整数。
2. 在函数内部，使用 `if` 语句来进行判断。如果 `n` 的值为 0 或 1，那么阶乘的值都是 1。函数直接返回 1。
3. 如果 n 大于 1，进入递归的情况。`return n * factorial(n - 1)` 目的是计算 `n` 乘以 `(n - 1)` 的阶乘，即 (n - 1)!。这里的递归是关键，它会继续调用 `factorial` 函数来计算 `(n - 1)` 的阶乘，然后将结果与 `n` 相乘，从而得到 `n!` 的值。并将结果返回。
4. `result = factorial(n)`：调用之前定义的 `factorial` 函数，传递输入的值 `n` 作为参数。并将其赋值给变量 `result`。

【练习】斐波那契数列

项目简介

斐波那契数列

知识模块

- Python 编程语言

知识点

- 分支语句-if
- 递归算法
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，使用递归算法，生成并输出斐波那契数列的前n项，其中n是用户指定的正整数。斐波那契数列，又称黄金分割数列，指的是：1、1、2、3、5、8、13、21、34....从第三个数开始，每个数字都是前两个数字之和。

解题思路

1. 当 n 小于等于 0 时，返回 0。
2. 当 n 等于 1 或 2 时，返回 1。
3. 否则，计算第 n 项斐波那契数列，即前两项的和。
4. 通过递归调用来计算第 n-1 项和第 n-2 项的和，直到 n 为 1 或 2。

完整代码

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# 计算斐波那契数列的第 10 项
n = 10
fib_result = fibonacci(n)
print(f"斐波那契数列的第 {n} 项为：{fib_result}")
```

代码讲解

1. `def fibonacci(n)`：这是定义一个名为 `fibonacci` 的函数，它接受一个参数 `n`，用于计算斐波那契数列的第 n 项。
2. `if n <= 0`：这是一个条件判断语句，用于检查 `n` 是否小于等于 0。
3. `return 0` 如果 `n` 小于等于 0，返回 0。
4. `elif n == 1 or n == 2`：这是另一个条件判断语句，检查 `n` 是否等于 1 或 2。
5. `return 1` 如果 `n` 等于 1 或 2，返回 1。
6. `return fibonacci(n - 1) + fibonacci(n - 2)` 这是递归的部分。当 `n` 大于 2 时，递归地调用 `fibonacci` 函数来计算第 `n-1` 项和第 `n-2` 项的和，然后返回结果。
7. `fib_result = fibonacci(n)` 调用 `fibonacci` 函数，传入参数 `n`，计算斐波那契数列的第 10 项。
8. `print(f"斐波那契数列的第 {n} 项为：{fib_result}")` 打印计算结果，显示斐波那契数列的第 10 项的值。

1.2 L2.Python面向对象编程

1.2.1 Python 高级语法

闭包与装饰器

闭包与装饰器

函数引用

讲解闭包之前，需要理解一个概念，Python 中定义的函数，也可以像变量一样，将一个函数名，赋值给另一个变量名，赋值后，此变量名就可以做为该函数的一个别名使用，进行调用函数，此功能在讲解列表操作的sort() 方法时使用过，sort()方法的 key 参数传入的就是一个函数名。

```
def show():
    print("Show Run ...")

show()
a = show
a()
```

注意: 在将一个函数名（函数引用）赋值给一个变量时，函数名后不能添加括号。

闭包

闭包（Closure）是指在一个嵌套的函数内部访问其外部函数中定义的变量或函数的能力。换句话说，闭包是一个函数对象，它可以记住并访问它创建时的上下文环境中的变量。

闭包通常由两个部分组成：内部函数和与其相关的环境变量。

- 内部函数是在外部函数中定义的函数，它可以访问外部函数中的局部变量和参数，以及外部函数所在的作用域中的变量。
- 环境变量是在外部函数中定义的变量或其他函数对象，它被内部函数引用并记住，即使外部函数执行完成后仍然存在。

闭包的特点包括：

1. 内部函数可以访问外部函数中定义的变量和参数，即使外部函数已经执行完毕。
2. 闭包可以在外部函数的作用域之外被调用和执行。
3. 闭包可以访问并修改外部函数中的局部变量，使其具有持久性。

闭包的应用场景包括：

1. 保护私有变量：可以使用闭包来创建私有变量和方法，通过内部函数的作用域和环境变量，可以实现对外部访问的限制。
2. 延迟执行：可以使用闭包来延迟某个函数的执行，即在函数外部创建一个闭包，将需要执行的函数作为内部函数，通过调用闭包来触发函数的执行。
3. 缓存数据：可以使用闭包来缓存一些昂贵的计算结果，以避免重复计算，提高程序的性能。

需要注意的是，在使用闭包时，要注意管理内存，避免产生不必要的内存泄漏问题。

```
def out_func():
    out_n = 100
    def inner_func():
        print(out_n)
    return inner_func


if __name__ == '__main__':
    of1 = out_func()
    of2 = out_func()

    of1()
    of2()
```

nonlocal

和全局变量一样，在函数内是不能直接修改函数外的变量的，如果修改全局变量需要使用 `global` 在函数内部声明变量为全局变量。 闭包中要修改变量也是一样，内函数是不能直接修改外函数中定义的变量的，如果需要修改，要在内函数中使用 `nonlocal` 关键字声明该变量为外函数的变量。

不使用 nonlocal 修饰

```
def out_func():
    out_n = 100
    def inner_func():
        out_n = 200
        print("inner:",out_n)
    print("outer1:",out_n)
    inner_func()
    print("outer2:",out_n)
    return inner_func


if __name__ == '__main__':
    of1 = out_func()
    of1()
# 结果：
# outer1: 100
# inner: 200
# outer2: 100
# inner: 200
```

使用 nonlocal 修饰

```
def out_func():
    out_n = 100
    def inner_func():
        nonlocal out_n
        out_n = 200
        print("inner:",out_n)
    print("outer1:",out_n)
    inner_func()
    print("outer2:",out_n)
    return inner_func


if __name__ == '__main__':
    of1 = out_func()
    of1()
# 结果：
# outer1: 100
# inner: 200
# outer2: 200
# inner: 200
```

装饰器

装饰器是Python提供的一种语法糖，装饰器使用 `@` 符号加上装饰器名称，用于修改其他函数的行为，并且在不修改原始函数定义和调用的情况下添加额外的功能。

装饰器提供了一种简洁而优雅的方式来扩展和修改函数或类的功能。它本质上就是一个闭包函数。

装饰器的功能特点:

- 不修改已有函数的源代码
- 不修改已有函数的调用方式
- 给已有函数增加额外的功能

装饰器的使用

由于装饰器本质上就是一个闭包函数，所以在使用自定义装饰器之前，需要先定义一个用来做为装饰器的闭包。

而闭包的外部函数名，就作为装饰器名使用。

```
import time

def count_time(func):
    def inner():
        start_time = time.time()
        func()
        stop_time = time.time()
        print(f'函数执行时间为{stop_time - start_time}秒')
    return inner

@count_time
def show():
    for i in range(3):
        print(f"第{i+1}次输出")
        time.sleep(1)

if __name__ == '__main__':
```


```
    show()

# 结果：
# 第 1 次输出
# 第 2 次输出
# 第 3 次输出
# 函数执行时间为3.0111730098724365秒
```

上面代码中，使用闭包实现了一个函数执行时间统计的功能。在show函数上，使用闭包函数做为装饰器为show 统计运行时间。

通过代码可以看出，在使用 count_time函数做为装饰器时，即没有改变show函数的内部定义，也没有改变show函数的调用方式，但却为show函数额外扩展了运行时间统计的功能，这就是装饰器的作用。

装饰器的本质

装饰器提供了一种简洁而优雅的方式（语法糖）来扩展和修改函数或类的功能。其本质就是函数的使用。

语**法糖**： 在计算机科学中，语法糖（Syntactic sugar）是指一种语法上的扩展，它并不改变编程语言的功能，只是提供了更便捷、更易读的写法，使得代码更加简洁和可理解。

常见的语法糖:

- 推导式
- 装饰器
- 切片
- 上下文管理器

Python解释器在遇到装饰器时，会将被装饰函数引用做为参数传递给闭包的外函数，外函数执行后，返回内函数的引用，此时，再将内函数引用赋值给被装饰器函数。

当Python解释器执行完装饰过程后，被装饰函数的函数名就不在保存原函数的引用，而是保存的闭包函数inner的引用。

而当执行被装饰函数时，实际执行的是闭包函数inner，由inner间接调用被装饰函数，完成整个调用过程。

```
@count_time
def show():
    pass
```

Python解释器解释过程：

```
show = count_time(show)
```

前面示例代码可修改为：

```
import time

def count_time(func):
    def inner():
        start_time = time.time()
        func()
        stop_time = time.time()
        print(f'函数执行时间为{stop_time - start_time}秒')
    return inner

def show():
    for i in range(3):
        print(f"第{i+1}次输出")
        time.sleep(1)

if __name__ == '__main__':
    show = count_time(show)
    show()
```

注意：装饰器装饰过程是由Python解释器执行，不需要显示书写，此处只为讲解原理演示。

通用装饰器

理论上，一个装饰器可以装饰任何函数，但实际前面定义的做为装饰器的 count_time 函数却只能装饰特定的无参无返回值的函数。

如果需要装饰器可以装饰任何函数，那么就需要解决被装饰函数的参数及返回值的问题。

可以通过可变参数和在内函数中返回被装饰函数执行结果的形式解决此问题。

```
# 做为装饰器名的外函数，使用参数接收被装饰函数的引用
def decorator(func):
    # 内函数的可变参数用来接收被装饰函数使用的参数
    def inner(*args, **kwargs):
        # 装饰器功能代码
        # 调用被装饰函数，并将接收的参数传递给被装饰函数，保存被装饰函数执行结果
        result = func(*args, **kwargs)
        # 返回被装饰函数执行结果
        return result
    # 返回内函数引用
    return inner
```

带参数装饰器

除了普通的装饰器使用方式外，在使用装饰器时，还需要向装饰器传递一些参数，比如测试框架 pytest 实现数据驱动时，可以将测试数据以装饰器参数形式传入，此时，前面定义的做为装饰器的闭包形式就不能满足需求了。

可以在通用装饰器外，再定义一层函数，用来接收装饰器的参数。

实现**代**码

```
def decorator_args(vars, datas):
    def decorator(func):
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        return inner
    return decorator

data = [(1,2,3),(4,5,6),(7,8,9)]
# 装饰器传参
@decorator_args("a,b,c", data)
def show(a,b,c):
    print(a,b,c)
```

装饰器传参原理

装饰器传参的本质就是链式语法的多次函数调用

`@decorator_args("a,b,c", data)` 解析

1. 先执行 `decorator_args("a,b,c", data)` 部分
2. 得到结果 `decorator` 与 `@` 结合变成装饰器形式 `@decorator`
3. 通过结果 `@decorator` 装饰器正常装饰被装饰函数

使用装饰器传参，实现数据驱动过程(了解)

此过程只用来讲解装饰器形式如何实现数据驱动过程，并没有完整实现。

```
# 接收装饰器参数的函数
# 参数一：以字符串形式接收被装饰函数的参数列表，需要与被装饰函数参数名保持一致，例："a,b,c"
# 参数二：以[(),(),()] 形式传入驱动数据。
def decorator_args(vars, datas):
    def decorator(func):
        # 将字符串参数分割备用
        v_keys = vars.split(",")
        # 定义保存 [{},{},{}] 形式的数据
        new_datas = []
        # 遍历数据，取出一组元素数据
        for item in datas:
            # 定义一个新字典，用来保存 变量名与传入数据组成的字典
            d_item = {}
            # 使用 zip 函数，同时遍历两个元组，变量名做为key，元素数据做为value
            for k, v in zip(v_keys, item):
                # 将 变量名和值对应保存到字典中
                d_item[k] = v
            # 将组合好的字典追加到新数据中备用
            new_datas.append(d_item)
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        # 遍历新数据，取出元素字典
        for item in new_datas:
            # 将字典中的数据解包传给内函数
            inner(**item)
        return inner
    return decorator

# 数据驱动数据
data = [(1,2,3),(4,5,6),(7,8,9)]
```

```
# 装饰器传参
@decorator_args("a,b,c", data)
def show(a,b,c):
    print(a,b,c)
```


文件操作

文件操作

文件操作是每一门编程语言中都必不可少的一项语法，通过文件，可以实现数据持久化存储，测试数据驱动文件的处理，程序配置文件的处理等。

在程序中操作文件和使用图形界面操作文件的过程基本一致，都要进行找到文件位置，打开文件，读写文件，关闭文件等操作。

打开文件

Python 使用 `open` 方法用于打开一个文件，并返回文件对象，在对文件进行处理过程都需要使用到这个函数，如果该文件无法被打开，会抛出 OSError。

完整格式：

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

简化格式：

```
open(file, mode='r', encoding=None)
```

- file: 必需，指定打开文件的路径（相对或者绝对路径）。
- mode: 可选，文件打开模式，默认为 `r` 只读模式
- encoding: 一般使用 `utf8`

mode参数常见下表：

字符	含意
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	以只写方式打开文件。如果该文件已存在则覆盖文件。如果该文件不存在则创建新文件。
a	以追加写入方式打开文件，如果文件存在则在末尾追加，文件不存在则创建新文件
b	二进制模式
t	文本模式（默认）

```
# 以写入文件打开 index.html 文件
file = open("index.html", "w")
```

文件关闭

文件在操作完以后，需要将其关闭，`close()` 方法用于关闭一个已打开的文件。

关闭后的文件不能再进行读写操作， 否则会触发 `ValueError` 错误。

`close()` 方法允许调用多次。

使用 `close()` 方法关闭文件是一个好的习惯。

格式：`fileObject.close()`

```
# 以写入文件打开 index.html 文件
file = open("index.html", "w")
# 关闭文件
file.close()
```

文件写入

write() 方法

`write()` 方法用于向文件中写入指定字符串。如果文件打开模式为 `b`，则要将字符串转换成 `bytes` 类型的二进制字符串,函数返回成功写入数据的长度。

格式：`fileObject.write( [ str ])`

```
# 以写入文件打开 index.html 文件
file = open("index.html", "w")
# 写入数据
file.write("<h1>文件写入标题</h1>")
file.write("\n")
file.write("<p>文件写入内容。。。。。</p>")
# 关闭文件
file.close()
```

writelines() 方法

`writelines()` 方法用于向文件中写入一序列的字符串。

这一序列字符串可以是由迭代对象产生的，如一个字符串列表。

注意：不要被方法名所迷惑，如果每个元素独占一行，需要在数据后指定换行符 `\n`。

格式：`fileObject.writelines(seq)`

```
datas = ["AAAAAAAAAAA\n","BBBBBBBBBBB\n","CCCCCCCCCCC\n","DDDDDDDDDDD\n"]
file = open('data.txt',"w")
file.writelines(datas)
file.close()
```

读取文件

read() 方法

`read()` 方法用于从文件读取指定的字节数，如果未给定或为负则读取所有。

格式：`fileObject.read([size=-1])`

```
file = open('data.txt',"r")
# 读取10个字符
content = file.read(10)
print(content)
# 读取所有内容
content = file.read()
print(content)
file.close()
```

readline() 方法

`readline()` 方法用于从文件读取整行，包括 `\n` 字符。如果指定了一个非负数的参数，则返回指定大小的字节数，包括 `\n` 字符。

格式：`fileObject.readline(size=-1)`

```
file = open('data.txt',"r")
# 读取10个字符
content = file.readline(10)
print(content)
# 读取文件指针所在行剩余所有内容
content = file.readline()
print(content)
file.close()
```

readlines()方法

`readlines()` 方法用于读取所有行(直到结束符 EOF)并返回列表。

格式：`fileObject.readlines()`

```
file = open('data.txt',"r")
# 以行为单位读取文件所有的内容
contents = file.readlines()
print(contents)
file.close()
```


错误分析与调试

调试与分析

在编写程序过程中，出现错误是再所难免的，出现错误后，如何能通过报错信息，快速找出并修复错误，是代码开发过程中非常重要的技能。

错误分析

程序出现错误并中断结束后，出现报错信息的错误都是不可怕的，大多都是一些语法性的错误，通过错误提示信息，就可以快速定位和解决错误。

```
1  data = [1,2,3,4,5]
2  print(data[0])
3  print(data[3])
4  print(data[5])
```

```
Run:  main
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3 /Users/liusuhui/Desktop/RecordCode/main.py
Traceback (most recent call last):
  File "/Users/liusuhui/Desktop/RecordCode/main.py", line 4, in <module>
    print(data[5])
IndexError: list index out of range
1
4

Process finished with exit code 1
```

错误分析:

1. 错误所在的文件
2. 错误所在的行
3. 错误出现的代码
4. 错误类型
5. 错误原因描述

有时候通过报错信息，并不能直接定位出错误，要通过报错信息，及程序的上下文逻辑来分析真正的报错原因。

main.py

```python
def getInputData():
    n = input("请输入一个数字:")
    msg = input("请输入一个数据:")
    return n,msg

def outputData(n,msg):
    for i in range(n):
        print(msg)

if __name__ == '__main__':
    n,msg = getInputData()
    outputData(n, msg)
```

if __name__ == '__main__'

Run: main

```
/Library/Frameworks/Python.framework/Versions/3.10/bin/python3 /Users/liusuhui/Desktop/RecordCode/main.py
请输入一个数字:5
请输入一个数据:Hello
Traceback (most recent call last):
  File "/Users/liusuhui/Desktop/RecordCode/main.py", line 12, in <module>
    outputData(n, msg)
  File "/Users/liusuhui/Desktop/RecordCode/main.py", line 7, in outputData
    for i in range(n):
TypeError: 'str' object cannot be interpreted as an integer

Process finished with exit code 1
```

在上面的错误中，共提示了两处报错位置，分别是12行和7行，这是代码追踪提示。

但真正的错误并不在这里，通过错误原因描述发现，是在执行range()函数时，并不能将字符串类型的参数解释成一个数字。

从代码实现逻辑上看，`output` 函数并没有任何问题，结合上下文的代码逻辑发现，问题出在 `getInputData` 函数中的数据获取，从键盘输入的所有内容都是以字符串形式保存到程序变量中，程序要求获得一个数字，那么在键盘输入后，应该使用强制类型转换，将输入数据转换成数字。

`n = input("请输入一个数字:")` 修改为 `n = int(input("请输入一个数字:"))`

print信息调试

很多人由于英文不太好，在查看报错信息时很吃力，也可以通过 `print()` 函数，自行输出信息来定位错误位置，虽然这种方式可以解决问题，但还是要慢慢学习，掌握如何查看报错提示。

示例代码:

```python
def getInputData():
    print("input run")
    n = input("请输入一个数字:")
    msg = input("请输入一个数据:")
    return n,msg

def outputData(n,msg):
    print("output run")
    for i in range(n):
        print("output forin run")
        print(msg)

if __name__ == '__main__':
    n,msg = getInputData()
    outputData(n, msg)
```

运行结果:

```
input run
请输入一个数字:1
请输入一个数据:1
output run
Traceback (most recent call last):
  File "/Users/liusuhui/Desktop/RecordCode/main.py", line 15, in <module>
    outputData(n, msg)
  File "/Users/liusuhui/Desktop/RecordCode/main.py", line 9, in outputData
    for i in range(n):
TypeError: 'str' object cannot be interpreted as an integer

Process finished with exit code 1
```

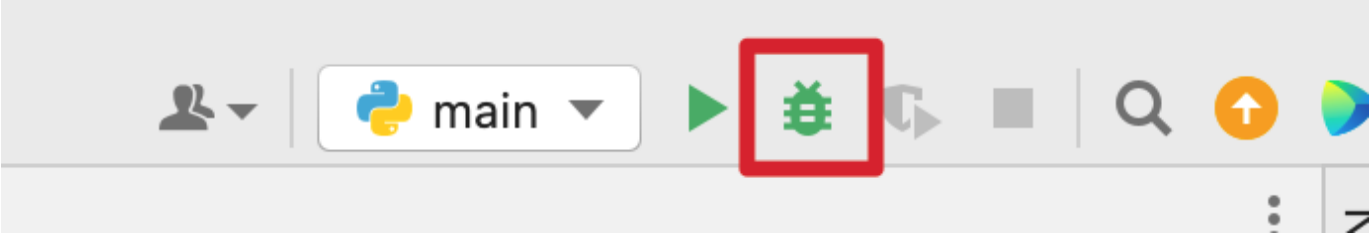
在示例代码中，加入了三条 `print()` 语句，用来输出一些信息，通过运行结果可以看出，`input run` 和 `output run` 都被正常输出，而 `output forin run` 没有输出，说明程序在输出该语句之前出现错误，此时就可以通过检查语法信息，上下文逻辑等来判断具体错误原因。

`print()` 语句在调试代码时，两条语句之间包含多少代码，视具体情况而定。不是必须在每条语句前后都加 `print()` 输出，在错误调试完成后，需要把输出注释或删除掉。

debug调试

除了前面两种方式外，还可以通过 PyCharm 的 `debug` 功能来调试程序，通过 `debug` 功能，还可以监控程序的执行过程。

在使用debug功能时，需要配合程序断点来进行调试。



程序断点

使用 PyCharm 编写代码时，可以在行号后通过点击添加删除断点。

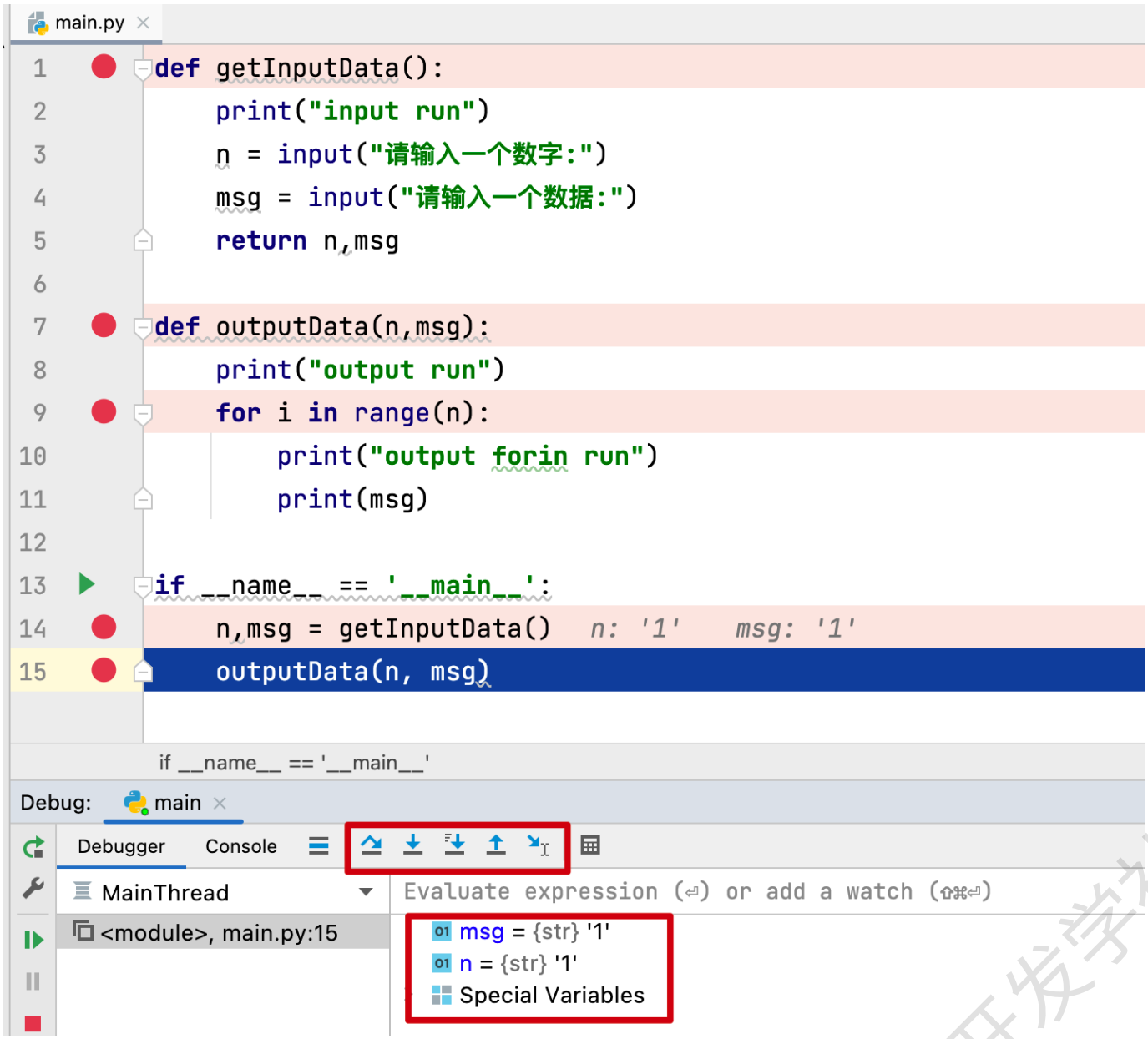
断点的作用是在debug调试程序时，遇到断点程序就会暂停执行，通过点击控制按钮，控制程序向下执行。

main.py

```python
def getInputData():
    print("input run")
    n = input("请输入一个数字:")
    msg = input("请输入一个数据:")
    return n,msg

def outputData(n,msg):
    print("output run")
    for i in range(n):
        print("output forin run")
        print(msg)

if __name__ == '__main__':
    n,msg = getInputData()
    outputData(n, msg)
```

调试控制

程序打好断点后，点击debug即可进入debug模式，程序遇到断点就会暂停执行，此时就需要通过控制按钮来控制程序的执行



横向按钮

- `Step Over`：步过按钮，将函数做为一条语句执行，不进入函数内部执行。
- `Step Into`：单步执行，会进入到函数内部逐条执行代码。
- `Step Into My Code`：单步执行，只进入自定义函数内部，不会进入系统函数内部。
- `Step Out`：步出按钮，跳出当前函数体，返回到此函数调用位置
- `Run to Cursor`：运行到光标处，当调试程序时，如果某一行没有打断点，又想暂停，可以将光标移动到目标行，点击该按钮
- `Evaluate Expression`：评估表达式，高级用法，可以在调试过程中查看程序的中间过程，比如查看参数 n 的类型。

竖向按钮

- `Rerun main`：重新运行 debug 功能
- `Modify Run Configuration`：修改运行配置
- `Resume Program`：继续执行，运行到下一断点处，如果没有，程序运行结束
- `Stop main`：停止 Debug
- `View Breakpoints`：显示程序中所有的断点。
- `Mute Breakpoints`：让所有断点失效，使用后所有断点为灰色，debug时，代码不会在断点处暂停。
- `Pin Tab`：钉住当前调试窗口标签，防止关闭。

异常处理

异常处理

编写程序时，即使语句或表达式使用了正确的语法，执行时仍可能触发错误。执行时检测到的错误称为**异常**，大多数异常不会被程序处理，程序会中断运行，并抛出异常信息。

如果不想发生异常时，程序被中断执行，可以编写程序处理选定的异常。

try-except

Python 使用 `try/except` 语句捕捉异常。

`try/except` 语句用来检测 `try` 语句块中的错误，从而让 `except` 语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在try里捕获它。

```
file = open("data.txt","r")
try:
    # 写入数据时可能会有问题
    file.write("写入的数据")
except IOError as err:
    print("文件不能写入", err)

file.close()
```

捕捉多个异常

如果一段代码可能会发生多种异常，并想在程序中都想处理，可以使用多个 `except` 分别捕捉异常。

可以捕捉 `Exception` 异常类型来处理所有的异常，如果有多个时，必须放在最后捕捉该异常，否则无法处理到其它异常。

```
file = open("data.txt","r")
try:
    # 写入数据时可能会有问题
    # file.write("写入的数据")
    # print(a)
    # print(3 / 0)
    # print([][10])
    print("hello" + 100)
except IOError as err:
    print("文件不能写入", err)
except NameError:
    print("标识符没有定义")
except ZeroDivisionError:
    print("除数不能为0")
except IndexError:
    print("下标越界了")
except Exception:
    print("程序运行出错，请检查代码")
file.close()
```

else 操作

Python 使用 `else` 在处理在代码无异常时的后续操作。

```
try:
    n = input("请输入一个数字:")
    num = int(n)
except Exception:
    print("元素无法转换为数字")
else:
    print("转换后成功",num)
```

finally 操作

Python 使用 `finally` 处理无论异常是否发异，都要执行的代码，一般用来完成清理工作。

```
try:
   file = open("data.txt","r")
   # file.write("A")
except Exception:
    print("文件操作报错")
finally:
    print("文件已关闭")
    file.close()
```

【练习】异常处理练习

项目简介

- 异常处理练习

知识模块

- Python 编程语言

知识点

- 异常处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写程序，让用户输入两个整数start和end,然后输出这两个整数之间的一个随机数。要求考虑用户输入不是整数的情况，以及start>end的情况。根据实际情况进行适当提示或输出。

解题思路

- 使用异常方法进行操作

完整代码

```
import random

try:
    # 提示用户输入起始值和结束值
    start = int(input("请输入一个整数作为起始值： "))
    end = int(input("请输入一个整数作为结束值： "))

    # 检查起始值和结束值的关系
    if start > end:
        raise ValueError("起始值不能大于结束值，请重新输入。")

    # 生成随机数并输出
    random_num = random.randint(start, end)
    print("随机数为：", random_num)

except ValueError as ve:
    # 捕获值错误异常并输出错误信息
    print(ve)

except Exception as e:
    # 捕获其他异常并输出通用的错误信息
    print("发生了一个错误：", e)
```

代码讲解

1. 导入 `random` 模块，用于生成随机数。
2. 提示用户输入起始值和结束值。
3. 使用 `int()` 函数将用户输入的字符串转换为整数类型，并赋值给变量 `start` 和 `end`。
4. 检查起始值和结束值的关系，如果起始值大于结束值，就抛出 `ValueError` 异常，同时提供错误提示消息。
5. 使用 `random.randint()` 函数生成一个在起始值和结束值之间的随机整数，并赋值给变量 `random_num`。
6. 打印随机数的值。
7. 使用 `try-except` 语句来捕获可能出现的异常。
8. 如果捕获到 `ValueError` 异常，说明用户输入的起始值大于结束值，输出相应的错误提示。
9. 如果捕获到其他类型的异常，输出通用的错误提示。

【练习】计算器

项目简介

计算器

知识模块

- Python 编程语言

知识点

- 异常处理
- 分支语句-if
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，可以执行加法、减法、乘法和除法操作。用户可以输入两个数字和运算符，然后计算并输出结果。实现计算器的功能（+、-、*、/），并处理异常情况，比如：输入的不是数字、除数为0等。

解题思路

1. 定义一个函数，接收两个数字和一个操作符，并返回计算结果。
2. 使用异常处理来捕获并处理可能发生的异常情况并输出错误消息。

完整代码

```
def calculate(num1, num2, operator):
    if operator == '+':
        return num1 + num2
    elif operator == '-':
        return num1 - num2
    elif operator == '*':
        return num1 * num2
    elif operator == '/':
        return num1 / num2
    else:
        return None

# 主程序
while True:
    try:
        # 获取用户输入的数值和运算符
        num1 = float(input("请输入第一个数："))
        operator = input("请输入运算符号（+、-、*、/）：")
        num2 = float(input("请输入第二个数："))

        # 调用calculate函数执行运算，并输出结果
        result = calculate(num1, num2, operator)
        if result is not None:
            print("计算结果为：", result)
        else:
            print("输入的运算符号不正确，请重新输入！")

        # 询问用户是否继续运算
        flag = input("是否继续运算？（Y/N）")
        if flag == 'N' or flag == 'n':
            break
    except ZeroDivisionError:
        print("错误：除数不能为零")
    except ValueError:
        print("错误：请输入有效的数字")
    except Exception as e:
        print("发生异常：",e)
    finally:
        print("程序结束。")
```

代码讲解

1. 代码

```
def calculate(num1, num2, operator):
    if operator == '+':
        return num1 + num2
    elif operator == '-':
        return num1 - num2
    elif operator == '*':
        return num1 * num2
    elif operator == '/':
        return num1 / num2
    else:
        return None
```

- 定义了一个名为 `calculate` 的函数，该函数接受两个数字和一个操作符作为参数。
- 根据操作符执行相应的运算，然后返回结果。如果操作符不是支持的运算符，返回 `None`

2. `while True`：定义了一个无限循环，意味着程序会一直执行循环内的代码，直到遇到 `break` 语句。
3. `try`：用于捕获可能发生的异常。
4. `result = calculate(num1, num2, operator)`:调用 `calculate ()` 函数，并传递参数，将返回的结果赋值给 `result`。
5. `except ZeroDivisionError`:用于捕获 `ZeroDivisionError` 异常，即除数为零的情况。
6. `except ValueError`:用于捕获 `ValueError` 异常，即用户输入无效的数字。
7. `except Exception as e`:用于捕获其他类型的异常。异常的详细信息会被存储在变量 `e` 中。
8. `finally`:不论是否发生异常，都会执行 finally 块中的代码。

【练习】计数器函数

项目简介

计数器函数

知识模块

- Python 编程语言

知识点

- 闭包与装饰器

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，实现一个计数器函数，该函数能够记录特定函数的调用次数。你需要使用闭包和装饰器来实现这个功能。

解题思路

1. 定义一个外部函数作为装饰器，它将创建一个闭包来保存计数器变量。
2. 在闭包内部，定义一个计数器变量并初始化为0。
3. 创建一个内部函数，用于实际执行被装饰的函数，并在其中将计数器递增。
4. 返回内部函数作为闭包的结果。
5. 将装饰器应用到需要计数的函数上。

完整代码

```python
# 定义装饰器函数
def count_calls(func):
    count = 0  # 初始化计数器变量

    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        print(f"函数 '{func.__name__}' 已被调用 {count} 次。")
        return func(*args, **kwargs)  # 调用原始函数

    return wrapper


# 使用装饰器来计数函数调用
@count_calls
def greet(name):
    return f"Hello, {name}!"


# 调用被装饰的函数
print(greet("Alice"))
print(greet("Bob"))
print(greet("Charlie"))
```

代码讲解

1. `def count_calls(func)`:定义了装饰器函数 `count_calls`，它接受一个函数 `func` 作为参数
2. `def wrapper(*args, **kwargs)`:定义内部函数 `wrapper`，它接受任意数量的位置参数 `*args` 和关键字参数 `**kwargs`
3. `nonlocal count`:在 `wrapper` 内部，使用 `nonlocal` 关键字来声明 `count` 变量，这使得闭包内部能够修改外部函数的变量。
4. `func.__name__`:用于获取函数的名称。
5. `return func(*args, **kwargs)`:调用原始函数 `func`，并将其结果返回。

【练习】读写文件

项目简介

读写文件

知识模块

- Python 编程语言

知识点

- 文件操作

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，将一些文本内容写入到文件中，并且能够从文件中读取内容并显示出来。

解题思路

- 使用文件的属性和方法进行操作

完整代码

```
file_name = "my_file.txt"
content = "Hello, this is a text file."

with open(file_name, "w") as file:
    file.write(content)
    print(f"内容已写入文件：{file_name}")

# 读取文件
with open(file_name, "r") as file:
    file_content = file.read()
    print(f"文件内容：\n{file_content}")
```

代码讲解

1. `content = "Hello, this is a text file."` 定义一个变量 content，表示要写入到文件的文本内容。
2. `with open(file_name, "w") as file`：- 使用 `open()` 函数以写入模式 `"w"` 打开文件，并将文件对象赋值给变量 `file`。`with` 语句保证在块结束后文件会自动关闭。
3. `file.write(content)` 使用文件对象的 `write()` 方法写入文本内容到文件。
4. `with open(file_name, "r") as file`：使用 `open()` 函数以读取模式 `"r"` 打开同一个文件，同样使用 `with` 语句以自动关闭文件。
5. `file_content = file.read()` 使用文件对象的 `read()` 方法读取文件内容，将内容赋值给变量 `file_content`。

1.2.2 Python 面向对象

面向对象概念

PYTHON面向对象的概念

面向过程编程

面向过程编程（Procedural Programming）是一种基于过程或函数的编程范式。

它将程序视为一系列的顺序执行的过程或函数，每个过程或函数完成特定的任务，通过调用其他过程或函数来协同工作。

在面向过程编程中，数据和函数（过程）是分离的，函数可以直接访问和操作数据。

在面向过程编程中，程序的执行流程通过顺序、条件判断、循环等语句来控制。常用的编程语言，如C、Fortran和Pascal，都是面向过程编程语言。

面向过程编程的特点包括：

1. **强调任务和步骤**：面向过程编程将程序划分为一系列的任务或步骤，这些任务按照顺序执行，以完成特定的功能。
2. **数据与函数分离**：在面向过程编程中，数据和函数（过程）是分离的，函数在需要时对数据进行操作。
3. **直接操作数据**：函数在执行过程中可以直接访问和操作数据，这样可能会导致数据在多个地方被访问和修改，增加了代码的复杂性和维护的困难。
4. **代码复用**：通过将常用的代码片段封装成函数，实现代码的复用。
5. 执**行效率**：由于没有面向对象编程中的对象创建、继承和多态等机制，面向过程编程通常可以更高效地执行。

面向过程编程适用于简单的任务和较小规模的项目，它注重问题的解决步骤和过程化的思维。它可以提供较好的性能，但对于复杂的系统和大型项目，面向对象编程更适合，因为面向对象编程更加灵活、可扩展和易于维护。

面向对象编程

面向对象编程（Object-Oriented Programming，OOP）是一种基于对象的编程范式。

它将程序视为一组相互作用的对象，每个对象都有自己的属性（数据）和方法（行为）。

通过对象之间的交互和消息传递，来完成任务和解决问题。

在面向对象编程中，将真实世界中的事物抽象成对象，并通过定义对象的类来描述对象的属性和方法。类是对象的模板，描述了对象共有的属性和可以执行的方法。通过创建类的实例（即对象），可以使用该实例的属性和方法。

面向对象编程的特点包括：

1. **封装性（Encapsulation）**：将数据和对数据的操作封装在对象中，仅向外部暴露必要的接口，隐藏了内部的实现细节。
2. 继**承性（Inheritance）**：通过继承机制，可以从已存在的类派生出新的类，新类自动获得了父类的属性和方法，并可以在此基础上进行扩展和修改。
3. **多态性（Polymorphism）**：同一种类型的对象在不同的上下文中可以有不同的行为，即在不同的情况下可以使用相同的接口来实现不同的功能。
4. **类与对象**：类是对象的模板，描述了对象的属性和方法。对象是类的实例，通过类创建的实例可以访问和操作该类定义的属性和方法。
5. **信息**传递：面向对象编程通过消息传递的方式实现对象之间的交互和通信。

面向对象编程具有代码模块化、可维护性、灵活性和可复用性的优点，它适用于复杂的系统和大规模的项目。常用的面向对象编程语言有Java、C++、Python等。使用面向对象编程可以更好地组织和抽象问题域，提高代码的可读性、可维护性和扩展性。

面向过程编程与面向对象编程的区别

面向过程编程（Procedural Programming）和面向对象编程（Object-Oriented Programming）是两种不同的编程范式，它们在思想、设计和实现上存在一些重要的区别。

1. **抽象程度**：面向过程编程将程序划分为一系列的过程或函数，通过函数之间的调用来完成任务。而面向对象编程将程序看作是一组相互作用的对象，每个对象都有其自己的属性和行为。
2. **封装性**：面向过程编程强调的是数据和函数的分离，函数可以直接访问和操作数据。而面向对象编程通过封装将数据和对数据的操作绑定在一起，只暴露出必要的接口，隐藏了内部的具体实现细节。
3. 继**承性**：面向对象编程具有继承的特性，可以通过继承机制从已存在的类派生出新的类，并继承父类的属性和方法，以实现代码的重用和扩展。而面向过程编程没有继承的概念，代码重用通常通过函数的封装和复用来实现。
4. **多**态**性**：面向对象编程支持多态，即同一种类型的对象在不同的上下文中可以有不同的行为。这使得代码更加灵活和可扩展。而面向过程编程没有多态的概念，需要通过条件语句来实现不同情况的处理。
5. 设计**思想**：面向对象编程注重的是问题领域的建模和抽象，关注对象之间的关系和交互。而面向过程编程更加注重问题的解决步骤和过程化的思维。

选择面向过程还是面向对象编程，取决于具体的需求和项目情况。面向对象编程更适合复杂的系统和大规模的项目，能够提高代码的模块化、可维护性和扩展性；而面向过程编程则更适合简单的任务和较小规模的项目，可以减少不必要的复杂性。

类和对象

类和对象

什么是类

在面向对象编程中，类（Class）是一种定义现实事物属性和方法的蓝图或模板。类描述了现实事物的特征（属性）和行为（方法）。可以把类看作是创建现实事物的原型。

类是现实事物的抽象，它定义了一类具有相似特征和行为的事物的通用结构和行为。类提供了对象所需的状态和行为，并定义了对象的初始化、操作和销毁等方法。

类由属性（也称为成员变量）和方法组成。属性是类的特征，用于描述类的状态。方法是定义在类中的函数，用于描述类的行为和操作。

通过使用类，我们可以创建多个具有相同属性和方法的对象。类中的属性和方法是相对独立的，每个对象都有自己的属性副本，但共享类中的方法。

例如，我们可以定义一个名为"Person"的类，该类可以有属性如姓名、年龄、性别等，方法如获取姓名、修改年龄、输出个人信息等。然后，我们可以创建多个实例（对象）来表示不同的人，每个实例都有自己的姓名、年龄和性别。

通过类的封装和抽象，可以更好地组织和管理代码，提高代码的可读性、可维护性和重用性。类是面向对象编程的重要概念之一。

什么是对象

在面向对象编程中，对象（Object）是类的一个实例。对象是具体存在的，具有状态和行为的实体。

可以把类看作是对象的模板或蓝图，描述了对象应该具有的属性和方法。当我们通过类创建一个具体的实例时，这个实例就是一个对象。

对象有两个关键的特征：状态和行为。

1. 状态（State）：对象的状态由它的属性（也称为成员变量）决定，属性表示了对象的特点和特征。例如，一个人对象的状态可以包括姓名、年龄、性别等属性，这些属性的值可以根据对象的实际情况而不同。
2. 行为（Behavior）：对象的行为由它的方法（也称为成员函数）决定，方法表示了对象能够执行的操作和行为。例如，一个人对象的行为可以包括走路、说话、工作等方法，通过调用这些方法可以让对象执行相应的行为。

对象是类的实例，可以根据类的定义创建多个对象。每个对象都有自己的属性值，在相同的类下，不同的对象可以具有不同的状态和行为。

通过对象，我们可以对数据进行封装和操作，通过对象之间的交互和消息传递，实现程序的逻辑和功能。

总结来说，对象是具体的实体，具有属性和方法。它是类的一个实例，通过类定义可以创建多个对象。对象是面向对象编程的核心和基本单位，通过对象的封装和抽象，我们可以更好地组织和管理代码。

类的定义

Python 使用 `class` 关键字来创建一个新类，`class` 之后为类的名称。

Python 存在一个根类 `object` 类，所有的类都由根类派生而来，如果自定义类继承于根类，可以省略。

格式：

```
class ClassName:
    pass
# 或
class ClassName(object):
    pass
```

示例：

```
class Plane(object):

    def flying(self,hour):
        print(f"飞机已飞行{hour}小时。。。")
```

实例对象

一个类定义好了，还不能使用，比如系统的list类，类只是规定了该类型的数据具有什么特征和行为，而真正要去使用这些特征和行为，必须有一个真实的列表存在。

实例对象本质上就是使用自定义类型去声明一个变量

格式： `实例对象名 = 类名()`

```
class Plane(object):

    def flying(self,hour):
        print(f"飞机已飞行{hour}小时。。。")

# 实例了两个对象
airPlane1 = Plane()
airPlane2 = Plane()
# 使用不同的对象调用了类中的方法
airPlane1.flying(3)
airPlane1.flying(5)
airPlane2.flying(3)
```


实例属性

实例属性

动态绑定

Python 中的属性变量都是使用动态绑定的方式绑定到实例对象上的。

格式： `实例对象名.实例属性名`

```
class Student:
    pass
# 实例对象
s1 = Student()
s2 = Student()

# 为实例对象s1动态绑定属性
s1.name = "Tom"
s1.age = 22
# 访问实例对象s1的属性
print(s1.name)
print(s1.age)

# 输出什么？
print(s2.name)
print(s2.age)
```

从代码中可以看出，在使用动态绑定属性时，给哪个实例对象绑定的属性，哪个对象才会拥有属性变量，没有绑定的则没有。

这显示是不符合面向对象思想的。

正常的实例属性定义在下一章节中讲解。

构造方法

构造方法

在上一章节中，通过动态绑定的方式为实例对象添加了属性。

但是这种操作显然是不符合逻辑的。每个实例对象一旦被实例，就应该含有类中定义的属性。

此时就需要使用构造方法来实现。

构造方法

构造方法 `__init__(self)` 在实例对象时自动调用，`self` 参数不需要手动传参，该参数在实例对象时，由解释器自动传入被实例的对象。

```
class Student:
    def __init__(self):
        print("Init Run ...")

s1 = Student()
s2 = Student()
```

此时，还不能定义属性，构造方法本质上就是一个具有特殊意义的函数，而在函数中直接定义变量不是属性，而是函数内的局部变量。

self

self 是一个特殊的关键字，用来表示当前被实例的对象，可以理解为人称代词**我**。

通过 `self` 可以定义或访问实例对象的属性或方法

格式：`self.属性名 = 值`

```
class Student:
    def __init__(self):
        self.name = "Tom"
        self.age = 22

s1 = Student()
s2 = Student()

print(s1.name)
print(s1.age)

print(s1.name)
print(s1.age)
```

通过 `构造方法` 和 `self` ,实现了为实例对象定义属性，其本质上还是为实例对象动态绑定属性，只是动态绑定的时机变了，从实例完对象再绑定属性，变成了在自动调用执行的构造方法中进行动态绑定。

并且，从上面的代码中还发现另外一个问题，就是所有实例的对象出来，都有共同的属性值，这显然也是不符合逻辑的。

带参构造方法

构造方法也可以携带参数，根据类中属性的定义，传入对应的参数对实例属性进行实始化。

格式：`__init__(self, args....)`

```
class Student:
    def __init__(self,name, age):
        self.name = name
        self.age = age

s1 = Student("Tom", 22)
s2 = Student("Jack", 23)

print(s1.name)
print(s1.age)

print(s1.name)
print(s1.age)
```

此时，才通过构造方法，实例了真正的对象。

`__str__(self)` 方法

在实例对象后，如果直接打印对象，输出该对象的相关信息，发现实际输出的并不是想要的结果。而是该实例对象的类型和地址。

如果想在输出实例对象时，按指定的格式输出，需要实现 `__str__(self)` 方法

该方法不接收除 `self` 以外的参数，`self` 参数自动传入，函数只能返回一个字符串。

```
class Student:
    def __init__(self,name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return f"Name: {self.name} -- Age: {self.age}"
s1 = Student("Tom", 22)
s2 = Student("Jack", 23)
print(s1)
print(s2)
```

实例方法

实例方法

实例方法用来定义对象的行为。实例方法本质上就是定义在类中的函数。

实例方法默认携带一个参数`self`，在程序执行时，由解释器自动传入调用该方法的实例对象，通过此参数，可以在当前实例方法中调用其它实例方法或属性。

格式：

```
def 方法名(self):
    pass

# 或

def 方法名(self, args....):
    pass
```

示例：

```
class Student:
    def __init__(self,name, age):
        self.name = name
        self.age = age
        self.courses = []
    def __str__(self):
        return f"Name: {self.name} -- Age: {self.age}"

    def select_course(self,courseName):
        self.courses.append(courseName)

    def all_course(self):
        print(f'{self.name}本学期选课如下：')
        for idx, c in enumerate(self.courses):
            print(f"第{idx+1}门课：{c}")

s1 = Student("Tom", 22)
s1.select_course("Python")
s1.select_course("Java")
s1.select_course("PyTest框架")
s1.all_course()
```

类属性

类属性

在 Python 中，一切皆为对象，类也不例外，在程序运行过程中，类也会做为一个对象使用。

类对象与实例对象不同，可以理解为实例对象是由类对象复制而来，每个实例对象之间具有数据独立性。而类对象在程序运行过程中，只有一个。

既然是对象，那么就可以拥有自己的属性，在类中定属性时，属性名有self前缀的是实例属性，而在类中直接定义的属性即为类属性。

```
# 定义一个饮水机类
class WaterDispenser:
    # 剩余水量
    surplus_water = 1500
    # 出水口
    def water_outlet(self, n):
        WaterDispenser.surplus_water -= n
        print("剩余水量：", WaterDispenser.surplus_water)

wd1 = WaterDispenser()
wd2 = WaterDispenser()

wd1.water_outlet(100)
print(wd1.surplus_water)
wd2.water_outlet(200)
print(wd2.surplus_water)
print(WaterDispenser.surplus_water)
```

类属性特征：

- 在类中直接定义的变量为类属性
- 在方法中使用类属性时，需要使用类名做为前缀 `类名.类属性名`
- 在类的外部可以通过类名或实例对象名访问类属性
- 所有的实例对象名共享一个类属性
- 实例对象只能获取类属性的值，不能直接进行修改，只能通过方法进行修改

类方法

类方法

除了类属性，类还有类方法。

同样，类方法也可以通过类名直接进行使用，类方法在定义时，需要使用 `@classmethod` 装饰器进行修饰。

与实例方法不同的是，实例方法有一个默认参数 `self`，代表当前调用方法的实例对象，而类方法的默认参数为 `cls`， 该参数也是在使用时，由解释器自动传入的，但传入的对象不是实例对象，而是类对象。

在类方法中，可以通过参数 `cls` 使用使用类属性。

一般类方法用来封装工具类使用，将一些复杂的代码逻辑封装成类方法，由类名直接调用，不需要实例对象，比如时间处理，网络请求处理等。

需要注意的是，如果类中即定义了实例属性，又定义了类方法，那么在类方法中是不能使用实例属性的，因为在使用类方法的过程中，实例对象不存在，所以不能使用实例属性。

示例： 封装了一个日期时间获取的工具类

```
import datetime
class Utils:
    now = datetime.datetime.now()

    @classmethod
    def current_date_time(cls):
        return cls.now

    @classmethod
    def current_date(cls):
        return cls.now.strftime("%Y-%m-%d")

    @classmethod
    def current_time(cls):
        return cls.now.strftime("%H-%M-%S")

    @classmethod
    def getYear(cls):
        return cls.now.year

    @classmethod
    def getMonth(cls):
        return cls.now.month

    @classmethod
    def getDay(cls):
        return cls.now.day


print(Utils.current_date_time())
print(Utils.current_date())
print(Utils.current_time())
print(Utils.getYear())
print(Utils.getMonth())
print(Utils.getDay())
```


静态方法

静态方法

除了类方法，Python 的类中还有一种静态方法。

静态方法在定义时，需要使用 `@staticmethod` 装饰器进行装饰，与类方法不同的是，静态方法没有默认参数。

静态方法和普通的函数本质上是一样的，只是定义在了类中。

一般情况下，静态方法同类方法一样，也是在封装工具类时使用，区别在于，静态方法中不需要使用类属性（不是不能使用，只是不建议）。

示例：封装两个数字操作的简单计算器

```
class Calc:
    @staticmethod
    def add(n1, n2):
        return n1 + n2

    @staticmethod
    def sub(n1, n2):
        return n1 - n2

    @staticmethod
    def mul(n1, n2):
        return n1 * n2

    @staticmethod
    def div(n1, n2):
        return n1 / n2


print(Calc.add(10, 20))
print(Calc.sub(10, 20))
print(Calc.mul(10, 20))
print(Calc.div(10, 20))
```

封装

封装

封装是面向对象编程中三大特征之一，指的是将数据和操作数据的方法打包在一起，形成一个类或对象。

封装的目的是隐藏对象的内部实现细节，提供一个安全且易于使用的接口，使得对象之间的交互更加简单和可靠。

封装主要包括以下几个方面的内容：

1. **数据隐藏**：通过将对象的数据属性设置为私有或受保护的，防止外部直接访问和修改对象的数据。这样可以确保对象的数据在被操作时不会被意外篡改或破坏。
2. **方法封装**：将对象对自身数据的操作封装在方法中，只通过方法来访问和修改对象的数据。这样可以确保对对象的操作符合预期，避免了外部错误地修改对象的数据。
3. **接口定义**：通过定义公共接口，将对象的功能暴露给外部使用者。使用者只需关心如何使用接口提供的方法，而不需要了解内部实现细节。这样可以提高代码的可读性和可维护性，同时也能够实现代码的模块化和复用。

封装的优势包括：

1. **数据安全性**：封装隐藏了对象的内部实现细节，保护了数据的完整性和安全性。外部无法直接访问或修改对象的数据，必须通过规定的方法进行操作，减少了意外错误的发生。
2. **代码模块化**：封装将对象的数据和操作打包在一起，实现了代码的模块化。不同的对象可以独立开发和测试，降低了代码的耦合性，增加了系统的可维护性和扩展性。
3. **简化接口**：封装将对象的功能通过公共接口暴露给外部，隐藏了内部实现细节。外部使用者只需了解接口的使用方法，而无需关心具体的实现。这降低了外部使用者的使用难度，也提高了代码的可读性和易用性。

访问控制

在Python中并没有像Java,C++ 一样，提供了 `public`，`protected`，`private` 这样的访问控制修饰符，Python 通过一种称为 `名称改写` 的方式，实现其它语言中访问控制修饰符的作用。

但是要注意的是，在Python中 `名称改写` 只是一种约定，并没有真正的实现私有的作用，在 Python 中只要想访问，所有的数据都可以拿到，获取方法在这里不讨论。

无下划线前缀(公有权限)

Python 中默认定义的属性和方法，都是公有的方法。无论是在类外，还是在派生的子类中，都可以进行访问，类似其它语言中的 `public` 修饰符的作用。

```
class A(object):
    def __init__(self):
        # 公有属性
        self.a = 10


    # 公有方法
    def show(self):
        # 在类中使用公有属性
        print(f"A: {self.a}")

obj = A()
# 在类外使用公有属性
print(obj.a)

# 在类外使用公有方法
obj.show()
```

_单下划线前缀（保护权限）

Python 在类中使用 `单下划线前缀` 实现其它语言中 `protected` 保护权限的功能，在属性或方法（包括类属性和类方法，作用相同）前添加一个单下划线，该属性或方法，在当前类中可以访问，在类外理论上不可访问（使用时不提示，但写出来程序可以运行，但有警告），在通过继承派生的子类中可以访问（继承在后面讲解）。

```
class A(object):
    def __init__(self):
        # 公有属性
```

```
        self.a = 10
        # 保护属性
        self._b = 20

    # 公有方法
    def show(self):
        # 在类中使用公有属性
        print(f"A: {self.a}")
        # 在类中使用保护属性
        print(f"B: {self._b}")
        # 在类中使用保护权限的方法
        self._display()

    # 保护权限的方法
    def _display(self):
        print(f"B: {self._b}")


obj = A()
# 在类外使用公有属性
print(obj.a)
# 在类外无法使用保护仅限的属性（不建议这样使用）
print(obj._b)
# print(obj.__c)
# 在类外使用公有方法
obj.show()
# 在类外无法使用保护权限的方法（不建议这样使用）
obj._display()
```

__ 双下划线前缀（私有属性）

Python 在类中使用 双下划线前缀 实现其它语言中 `private` 私有权限的功能，在属性或方法（包括类属性和类方法，作用相同）前添加一个双下划线，该属性或方法，只能在当前类中可以访问，在类外任何位置不可访问（只是理论上不可访问，通过某些方式，还是可以在类外访问，不建议这样使用）。

```
class A(object):
    def __init__(self):
        # 公有属性
        self.a = 10
        # 保护属性
        self._b = 20
        # 私有属性
        self.__c = 30

    # 公有方法
    def show(self):
        # 在类中使用公有属性
        print(f"A: {self.a}")
        # 在类中使用保护属性
        print(f"B: {self._b}")
        # 在类中使用私有属性
        print(f"C: {self.__c}")
        # 在类中使用保护权限的方法
        self._display()
        # 在类中使用私有方法
        self.__info()


    # 保护权限的方法
    def _display(self):
        print(f"B: {self._b}")

    # 私有权限的方法
    def __info(self):
        # 在类中使用私有属性
        print(self.__c)
obj = A()
# 在类外使用公有属性
print(obj.a)
# 在类外无法使用保护仅限的属性（不建议这样使用）
print(obj._b)
# 在类外使用私有属性，访问失败
# print(obj.__c)
# 在类外使用公有方法
obj.show()
# 在类外无法使用保护权限的方法（不建议这样使用）
obj._display()
# 在类外访问私有方法，访问失败
# obj.__info()
```

双下划线前缀与后缀

在 Python 中还有一种同时具有前后双下划线的变量或方法，这些方法是 Python 中的魔法属性或魔法方法，这些属性或方法名被赋予了特殊的作用，比如：初始化方法 `__init__()`，对象描述方法 `__str__()`，还有 `__iter__()`，`__next__()` 方法等，这些都是魔法方法。

计算属性

计算属性

在封装类时，如果实例属性定义为公有属性，在使用过程中不安全，一般建议定义为私有属性，但是私有属性又不能在类的外部直接访问，此时，需要为私有实例属性提供访问的操作接口方法。

属性访问器和修改器

属性访问器：也称为 `Getter` 方法，用来返回某个属性的值，该方法无参数，但必须有返回值，一般以 `get_xxx` 命名。 属性修改器：也称为 `Setter` 方法，用来给某个属性值进行修改，该方法接收一个参数，且无返回值，一般以 `set_xxx` 命名。

```
class Person:
    def __init__(self, name):
        self._name = name

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name


if __name__ == '__main__':
    tom = Person("tom")
    print(tom.get_name())
    tom.set_name("Tom")
    print(tom.get_name())
```

虽然实现了需求，但此方式在使用实例属性时，变成了以方法形式使用，不够简洁。

Property 装饰器

Python 提供了一个名为 `property` 的装饰器，通过该装饰器可以为私有属性提供访问器和修改器方法，但在使用时，依然可以以属性的形式进行使用。

- 提供数据访问功能（`getter`）
- 计算属性
- 语法：使用 `@property` 装饰器
- 调用：`实例.方法名`

- 提供数据操作功能（`setter`）
- 语法：使用 `@计算属性名.setter` 装饰器
- 调用：`实例.方法名`

注意：

- 当定义计算属性时，必须先使用 `property` 装饰器定义计算属性的访问器访问，然后才可以利用计算属性名定义修改器方法。
- 访问器方法和修改器方法不需要再使用 `get` 和 `set` 做为前缀。

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def username(self):
        return self._name

    @username.setter
    def username(self, name):
        if name.isalpha():
            self._name = name


if __name__ == '__main__':
    tom = Person("tom")
    print(tom.username)
    tom.username = "Tom"
    print(tom.username)
```

计算属性的优势

- 可以隐藏实现细节
- 可以进行访问控制
- 可以进行数据验证
- 不改变属性操作方式基础上完成复杂的逻辑控制

继承

继承

继承是面向对象编程中的三大概念之二，指的是一个类基于另一个类来创建。

创建出来的新类称为子类或派生类。被继承的类称为父类或基类。

通过继承，子类可以继承父类的属性和方法，并且可以在此基础上添加新的属性和方法，或者对继承的属性和方法进行修改。

继承的主要特点包括：

1. **继承关系**：继承创建了一个父类和子类之间的关系。子类继承了父类的特性，包括属性和方法。子类可以重用父类的代码，减少了代码的冗余。
2. **子类的扩展**：子类可以在继承父类的基础上，添加新的属性和方法。这样可以对父类进行扩展，使得子类具有更多的功能。
3. **代码共享和重用**：通过继承，子类可以共享父类的代码。父类中通用的属性和方法可以被多个子类继承和使用，提高了代码的重用性，并减少了开发时间和成本。
4. **继承的层次结构**：继承可以形成一个层次结构，其中一个父类可以有多个子类，而子类又可以成为其他子类的父类。这种层次结构可以更好地组织和管理代码，使得代码更加结构化和模块化。

继承的优势包括：

1. **代码重用**：继承允许子类重用父类的代码，减少了代码的冗余，提高了代码的可维护性和复用性。
2. **扩展性**：通过继承，子类可以在父类的基础上添加新的属性和方法，实现对父类的扩展，使得子类具有更多的功能。
3. **类型的兼容性**：由于子类继承了父类的特性，子类可以被当作父类的实例来使用。这样，在需要父类类型的地方，可以使用子类的实例，增加了代码的灵活性和可扩展性。

需要注意的是，虽然继承可以提供代码重用和扩展的好处，但过度使用继承可能导致代码的复杂性和耦合性增加。因此，在设计代码时，应该合理使用继承，并遵循单一责任原则和开闭原则，保持代码的简洁和灵活。

单继承

单继承是指一个子类只继承一个父类。

```
class A(object):
    # A 继承自 object 根类
    def show(self):
        print("父类A的方法")

class B(A):
    # B类 继承自 A类
    def display(self):
        print("子类B的方法")

b = B()
# 子类对象使用自己的方法
b.display()
# 子类对象使用父类的方法，如果父类有没有该方法则继续向上查找，直到根类
b.show()
```

方法重写

在子类中，可以对父类中的方法实现进行重写，实现新的功能实现。

```
class A(object):
    # A 继承自 object 根类
    def show(self):
        print("父类A的方法")

class B(A):
    # 子类重写父类方法
    def show(self):
        print("子类B的方法")

b = B()
# 当子类方法与父类方法同名时，调用子类方法
b.show()
```

super()

如果在子类中还要使用父类中的方法，可以使用 `super()` 函数来调用父类中的方法。

比如在重写父类方法时，还要保留父类方法的功能。

```
class A(object):
    # A 继承自 object 根类
    def show(self):
        print("父类A的方法")

class B(A):
    # 子类重写父类方法
    def show(self):
        # 使用 super() 调用父类方法
        super().show()
        print("子类B的方法")

b = B()
# 当子类方法与父类方法同名时，调用子类方法
b.show()
```

单继承的初始化

在子类对象初始化时，需要给出父类初始化所需的参数，然后使用 `super()` 调用父类初始化方法去初始化父类的属性。

```
class A(object):
    # A 继承自 object 根类
    def __init__(self, a):
        self.a = a


class B(A):
    def __init__(self, a, b):
        super().__init__(a)
        self.b = b

b = B("A","B")
print(b.a)
print(b.b)
```

继承的访问控制

无论在方法的重写，还是初始化时，父类的工作就让父类自己去完成，子类只负责自己部分的实现。

比如：如果在初始化时，想在子类中初始化父类的一个私有属性，这是不能实现的，但是可以调用父类的初始化方法对私有属性进行初始化

```
class A(object):
    # A 继承自 object 根类
    def __init__(self, a, b, c):
        self.a = a
        self._b = b
        self.__c = c

    def show(self):
        print(f"A: {self.a}")
        print(f"B: {self._b}")
        print(f"C: {self.__c}")


class B(A):
    def __init__(self,a,b,c,d):
        super().__init__(a,b,c)
        self.d = d

    def show(self):
        super().show()
        print(f"D: {self.d}")

b = B(1,2,3,4)
b.show()
```

多继承

多继承是指一个子类可以同时继承多个父类，此时子类同时拥有多个父类中的属性和方法

```
class FA(object):
    def fa_show(self):
        print("FA Show Run...")

class FB(object):
    def fb_show(self):
        print("FB Show Run...")
```

```
class S(FA, FB):
    def s_show(self):
        print("S Show Run...")


s = S()
s.s_show()
s.fa_show()
s.fb_show()
```

多继承同名方法查找顺序

如果在一个子类所继承的多个父类中，具有同名方法，那么在调用该方法名的方法时，Python 会使用C3算法实现的 MRO（方法解析顺序）顺序来确定查找的先后顺序，一般情况可以理解成是按继承类的书写顺序。

```
class FA(object):
    def show(self):
        print("FA Show Run...")

class FB(object):
    def show(self):
        print("FB Show Run...")

class S(FB, FA):
    def s_show(self):
        print("S Show Run...")


s = S()
s.s_show()
s.show()
```

多继承初始化

在多继承中，由于有多个父类，每个父类的属性都需要单独初始化，这时 `super()` 函数只能引用继承书写顺序上的第一个父类，其它的父类是无法通过 `super()` 引用的，所以也就无法利用 `super()` 函数进行初始化。

此时，可以使用直接指定父类名的方式调用该父类中的方法。

此方法也适用于多继承中的方法重写。

```
class FA(object):
    def __init__(self, a):
        self.a = a

class FB(object):
    def __init__(self, b):
        self.b = b

class S(FB, FA):
    def __init__(self, a, b, c):
        FA.__init__(self, a)
        FB.__init__(self, b)
        self.c = c


c = S(1,2,3)
print(c.a)
print(c.b)
print(c.c)
```

多态

多态

多态是面向对象编程中三大概念之三，它允许不同的对象对同一个消息作出不同的响应。

简单来说，多态是指同一个方法或操作符在不同的对象实例上可以有不同的行为。这意味着可以通过一个共同的接口或基类引用不同的子类对象，并根据实际的对象类型来调用相应的方法。

多态性在实际应用中提供了很多好处，包括：

1. 简化代码：通过以相同的方式处理不同的对象，并使用统一的接口进行编程，可以降低代码的复杂性和重复性。
2. 可维护性：多态可以提高代码的可维护性。当需要新增一种子类时，不需要修改已有的代码，只需要创建一个新的子类并继承父类，就能够在原有的代码基础上实现新的功能。
3. 扩展性：由于多态允许在不修改已有的代码的情况下新增功能，因此可以更容易地对系统进行扩展和适应需求的变化。

多态性的实现通常通过继承和方法重写来实现。在继承关系中，子类可以重写父类的方法，在父类引用子类对象时，调用的实际上是子类重写后的方法。

```
# 中医
class Father:
    def cure(self):
        print("使用中医方法进行治疗。。。")

# 西医
class Son(Father):
    def cure(self):
        print("使用西医方法进行治疗。。。")


# 患者
class Patient:
    def needDoctor(self, doctor):
        doctor.cure()


if __name__ == '__main__':
    oldDoctor = Father()
    littleDoctor = Son()

    patient = Patient()

    patient.needDoctor(oldDoctor)
    patient.needDoctor(littleDoctor)
```

鸭子类型

鸭子类型（Duck Typing）是一种动态类型的概念，它源自于“走起来像鸭子、叫声像鸭子、看起来像鸭子，那么它就是鸭子”的观念。

在鸭子类型中，一个对象的适用性不是由它的类或接口决定，而是由它的方法和属性是否与所需的方法和属性匹配来决定。换句话说，只要一个对象具有特定方法和属性，我们就可以将其视为具有相同类型。

举个例子，如果我们需要一个能“叫”的对象，并且某个对象有一个名为 `quack()` 的方法，那么我们可以将该对象视为一个“鸭子”，不管它实际上是什么类的对象。换句话说，我们关注的是对象的行为而不是其类型。

鸭子类型在动态语言中特别常见，比如 Python。在 Python 中，不需要显式地继承或实现接口，只要一个对象具有必需的方法和属性，它就可以被认为是某种类型。这使得 Python 具有灵活性和简洁性，可以更自由地处理不同类型的对象。

```
# 中医
class Father:
    def cure(self):
        print("使用中医方法进行治疗。。。")

# 西医
class Son(Father):
    def cure(self):
        print("使用西医方法进行治疗。。。")

# 兽医
class AnimalDoctor:
    def cure(self):
        print("使用兽医方法进行治疗。。。")

# 患者
class Patient:
```

```
    def needDoctor(self, doctor):
        doctor.cure()


if __name__ == '__main__':
    oldDoctor = Father()
    littleDoctor = Son()
    animalDoctor = AnimalDoctor()

    patient = Patient()

    patient.needDoctor(oldDoctor)
    patient.needDoctor(littleDoctor)
    patient.needDoctor(animalDoctor)
```

鸭子类型通常是动态语言的特性，相比于静态类型语言，它在编译时没有类型检查。这意味着无法在编译阶段对类型不匹配或缺失方法和属性进行检测，可能会导致运行时错误。

类型检查

Python 中提供了 `isinstance()` 和 `issubclass()` 两个函数，用来对数据进行检查判断。

isinstance()

Python 中使用 `isinstance()` 来检查一个实例的类型

格式： `isinstance(obj, type)`

判断 obj 对象是否是 Type 指定类型或其父类类型的实例。

```
    print(isinstance(littleDoctor, Father))
    print(isinstance(littleDoctor, Son))
    print(isinstance(littleDoctor, AnimalDoctor))
```

前面示例的代码可以进行优化：

```
# 患者
class Patient:
    def needDoctor(self, doctor):
        if isinstance(doctor, Father):
            doctor.cure()
        else:
            print("此大夫医疗方法不适用病人。。。")
```

issubclass()

Python 中还可以使用 `issubclass()` 来检查类的继承关系。

格式： `issubclass(Type1, Type2)`

判断 Type1 是否是 Type2 的子类

```
    print(issubclass(Father, Father))
    print(issubclass(Son, Father))
    print(issubclass(AnimalDoctor, Father))
```

前面的示例也可优化为：

```
class Patient:
    def needDoctor(self, doctor):
        if issubclass(doctor.__class__, Father):
            doctor.cure()
        else:
            print("此大夫医疗方法不适用病人。。。")
```

`__class__` 是一个魔法属性，用来获取当前实例对象的类。

类型注解

类型注解

Python 是一种动态类型语言，变量的类型是程序在运行时通过保存的数据动态推导得到，该特性使得在开发过程中不必过度关注变量的类型。

但随着项目越来越大，代码也就会越来越多，在这种情况下，很容易不记得某一个方法的入参类型是什么，一旦传入了错误类型的参数，再加上python是解释性语言，只有运行时候才能发现问题， 这对大型项目来说是一个巨大的灾难。

在Python 3.5版本中，引进了一种新的语法来给函数或变量增加注释，即类型注解。

Python 类型注解是一种可选的静态类型检查机制，它在注释中标注变量的类型，以提高程序的可读性和可维护性。

类型注解的特征如下：

- 类型检查，防止运行时出现参数和返回值类型、变量类型不符合。
- 作为开发文档附加说明，方便使用者调用时传入和返回参数类型。
- 在函数参数中使用时，形参变量会根据类型进行相应的代码提示。
- PyCharm 目前支持 typing 检查，参数类型错误会黄色提示。
- 加入类型注解后并不会影响程序的运行，不会报正式的错误，只有提醒。

类型注解基本使用

Python 类型注解采用冒号 `:` 后跟预期类型的方式进行标注。

```
num: int = 10
print(num)
```

该代码的作用是定义 `num` 变量，并通过类型注解约定 `num` 变量需要接收 `int` 类型的数据。

但类型注解只会对代码进行提醒，并不会报错误提示。

```
num: int = 10
print(num)
num = 20
print(num)
num = "hello"
print(num)
```

该代码中，约定 `num` 接收 `int` 类型数据，前两次赋值没有问题，但第三次赋值时，类型不符，IDE 会以黄色背景进行提示，内容为 `Expected type 'int', got 'str' instead` ，但这并不是错误提示，该代码依然可以正常执行。

基本类型注解

除了在变量定义时使用类型注释，更多的情况下是在函数的参数与返回值中使用注释注解，实现上下文的关联。

```
def show(n: int, msg: str) -> None:
    for i in range(n):
        print(msg.upper())

show(3, "hello")
```

- `n: int` ：指定了输入参数 `n` 为 `int` 类型，
- `msg: str` ：指定了输入参数 `msg` 为 `str` 类型，
- `-> None` ：指定了 `show` 函数的返回值为 `None` 类型，即无返回值。

在调用函数书写参数时，IDE 会根据函数中的类型注解对函数进行参数及类型的提示。

容器类型注解

tuple 类型注解 元组基本使用

```
def show(data : tuple) -> None:
    for d in data:
        print(d)
```

```
show((1, 2, 3))
show(('a', 'b', 'c'))
show((1, 2, 3, 'a', 'b', 'c'))
```

指定元组中元素类型

```
def show(data : tuple[int]) -> None:
    for d in data:
        print(d)

show((1,))
show((1, 2, 3)) # 不符合注解类型
show(('a', 'b', 'c')) # 不符合注解类型
show((1, 2, 3, 'a', 'b', 'c')) # 不符合注解类型
```

由于元组的不可变特殊性，当指定了元素类型，随之也指定了无组的元素个数，代码中约定传入具有一个整型数据元素的元组。

指定元组中多个不同类型的元素

```
def show(data : tuple[int, str, bool]) -> None:
    for d in data:
        print(d)

show((1,"hello", True)) # 多个类型数据要一一对应
show((1, 2, 3)) # 不符合注解类型
show(('a', 'b', 'c')) # 不符合注解类型
```

指定元组中任意个不同类型的元素

使用任意类型时，需要使用 `typing` 模块中的 `Any` 类型，可变数量使用 `...` 表示。

```
from typing import Any

def show(data: tuple[Any, ...]) -> None:
    for d in data:
        print(d)


show((1, "hello", True, 123))
show((1, 2, 3))
show(('a', 'b', 'c'))
```

list 类型注解

由于列表的可变特性，使用相对简单。

列表基本使用

```
def show(data: list) -> None:
    for d in data:
        print(d)


show([1, 2, 3])
show(["a", 'b', 'c'])
show([1, 2, 3, "a", 'b', 'c'])
```

列表中指定元素类型

```
def show(data: list[int]) -> None:
    for d in data:
        print(d)


show([1, 2, 3])
show(["a", 'b', 'c']) # 不符合注解类型
show([1, 2, 3, "a", 'b', 'c'])
```

在指定元素中元素类型时，实参列表中只要存在注解类型数据，即使包含了其它类型也认为符合类型要求。

dict 类型注解

字典与列表类似，具有元素个数可变性，所以只需指定 `key` 与 `value` 的类型即可。

```
def show(data: dict[str,int]) -> None:
    for d in data:
        print(d)
```

```
show({"a": 97})
show({"a": 97, "b": 98})
show({"c": "CC"})  # 不符合注解类型
```

Union 类型注解

可以使用 `typing` 模块中的 `Union` 指定多个类型

```
from typing import Union


def show(data: Union[str,int,float,bool]) -> None:
    print(data)


show(1)
show("a")
show(3.14)
show(True)
```

Sequence 类型注解

可以使用 `typing` 模块中的 `Sequence` 指定任意可迭代类型。

`Sequence` 所提示的是任何可以被索引的数据: 列表,元祖,字符串等。

注意：在定义函数时 `[]` 中只可以指定一个数据类型，意味着在传入的列表或者元组等可索引数据时，内部元素的数据类型需要是统一的。

```
from typing import Sequence

def show(data: Sequence[int]) -> None:
    print(data)


show((1,2,3))
show([1,2,3])
show("123")
```

Optional 类型注解

当函数的参数有默认值，导致参数不是必须要传入的，那么你可以尝试使用 `typing` 模块中的 `Optional` 来做到类型提示。

```
from typing import Optional


def show(data: Optional[int] = 0) -> None:
    print(data)


show()
show(1)
```

Callable 类型注解

当需要表明某个函数的参数是函数时可以使用 `typing` 模块中的 `Callable` 作为类型提示。

```
from typing import Callable


def show(data: int) -> None:
    print(data)

def callback(func: Callable, data: int)-> None:
    func(data)

callback(show, 1)
```

自定义类作为类型注解

自定义类可以直接做为类型进行注解使用。

```
class Person:
    pass

def show(obj: Person)->None:
    print(obj)

show(Person())
p = Person()
show(p)
```

【练习】封装网络请求工具

项目简介

封装网络请求工具

知识模块

- Python 编程语言

知识点

- 类和方法
- HTTP 请求
- 请求头
- 回调函数

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，封装一个网络请求工具类。

```
对 get/post/put/delete四种请求方式分别封装对应的请求方法
参数有 `url, header(使用字典)`, `callback`回调函数
`callback`回调整函数中，用来显示前两个参数的信息
```

解题思路

1. 根据需求确定封装的级别
2. 导入相应的库，`requests` 库。
3. 定义工具类
4. 编写请求方法
5. 设置请求头部信息
6. 处理回调

完整代码

```python
class RequestTools(object):
    @classmethod
    def get(cls, url, header, callback):
        callback(url, header)

    @classmethod
    def post(cls, url, header, callback):
        callback(url, header)

    @classmethod
    def put(cls, url, header, callback):
        callback(url, header)

    @classmethod
    def delete(cls, url, header, callback):
        callback(url, header)


def func(url, header):
    print(f"请求的网址是{url}")
    print(f"请求的信息有：")
    for i in header:
        print(i,header[i])

if __name__ == '__main__':

    RequestTools.get('http://www.baidu.com', {"content-type":"text/html"}, func)
    RequestTools.post('http://www.baidu.com', {"content-type":"text/html"}, func)
```

```
    RequestTools.put('http://www.baidu.com', {"content-type":"text/html"}, func)
    RequestTools.delete('http://www.baidu.com', {"content-type":"text/html"}, func)
```

代码讲解

1. `get` 方法接受三个参数：`url`（请求的网址）、`header`（请求头部信息）和 `callback`（回调函数）。在这个方法内部，我们直接调用了传入的 `callback` 函数，并将 `url` 和 `header` 作为参数传递给它。
2. `post`、`put` 和 `delete` 方法与 `get` 方法的逻辑相同，都接受相同的参数并调用传入的 `callback` 函数。
3. `func` 是一个回调函数，它接受两个参数 `url` 和 `header`。在这个回调函数中，我们打印了传入的 `url` 并遍历了 `header` 字典，打印了字典中的键和对应的值。

【练习】动物园

项目简介

动物园系统

知识模块

- Python 编程语言

知识点

- 实例方法
- 实例属性
- 类属性
- 构造方法
- 封装
- 继承
- 多态

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

设计一个简单的动物园系统，其中包含不同类型的动物（如狗、猫和鸟）。每个动物都有自己的属性（如名字、年龄）和行为（如发出声音）。使用封装、继承和多态来完成。

解题思路

1. 创建一个动物基类，其中包含一些共同的属性（如名字、年龄）和方法（如发出声音）。定义构造函数 **init** 和发声方法。
2. 创建子类（每种动物），分别继承自基类。
3. 在每个子类中，可以定义不同的属性和重写发声方法。
4. 创建函数来实例化不同类型的动物，并调用它们的方法。

完整代码

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def make_sound(self):
        pass


# 定义 Dog 子类
class Dog(Animal):
    def __init__(self, name, age):
        super().__init__(name, age)

    def make_sound(self):
        print(f"{self.name} 发出汪汪叫声！")


# 定义 Cat 子类
class Cat(Animal):
    def __init__(self, name, age):
        super().__init__(name, age)

    def make_sound(self):
        print(f"{self.name} 发出喵喵叫声！")


# 定义 Bird 子类
class Bird(Animal):
    def __init__(self, name, age):
        super().__init__(name, age)
```

```
    def make_sound(self):
        print(f"{self.name} 发出鸟叫声！")


# 创建动物对象并调用方法
dog = Dog("狗狗", 3)
cat = Cat("猫咪", 2)
bird = Bird("小鸟", 1)

dog.make_sound()
cat.make_sound()
bird.make_sound()
```

代码讲解

1. 定义父类

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def make_sound(self):
        pass
```

- 这部分定义了一个基类 `Animal`，其中包含了构造函数 `__init__`，该构造函数接收动物的名字和年龄作为参数，并将它们存储为对象属性。还定义了一个未实现的 `make_sound` 方法，它在基类中只是一个占位符方法。

2. 定义子类

```
class Dog(Animal):
    def __init__(self, name, age):
        super().__init__(name, age)

    def make_sound(self):
        print(f"{self.name} 发出汪汪叫声！")
```

- 这部分定义了 `Dog` 子类，它继承自基类 `Animal`。它的构造函数使用 `super().__init__` 来调用基类的构造函数，从而初始化继承的属性。`make_sound` 方法被重写，实现了狗叫的功能。

- `Cat` 子类和 `Bird` 子类与 `Dog` 子类类似

3. 实例对象

```
dog = Dog("狗狗", 3)
cat = Cat("猫咪", 2)
bird = Bird("小鸟", 1)

dog.make_sound()
cat.make_sound()
bird.make_sound()
```

- 实例化了不同类型的动物对象，即 `dog`、`cat` 和 `bird`。然后，调用了它们的 `make_sound` 方法，由于多态性的存在，实际上调用了每个子类中不同的 `make_sound` 方法，分别输出不同的叫声。

【练习】房子家具管理系统

项目简介

房屋家具管理系统

知识模块

- Python 编程语言

知识点

- 对象的封装
- 类的构造方法和实例属性
- for循环

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序： - 1.房子有户型，总面积和家具名称列表，新房子没有任何的家具。 - 2.家具有名字和占地面积，其中 - 床：占4平米 - 衣柜：占2平米 - 餐桌：占1.5平米 - 3.将以上三件家具添加到房子中 - 4.打印房子时，要求输出:户型，总面积，剩余面积，家具名称列表

解题思路

1. 首先创建一个房子类（House），包括房子的户型和总面积属性，以及一个家具列表属性。
2. 在房子类中定义一个添加家具的方法（add_furniture），该方法接收一个家具对象作为参数。
3. 在添加家具的方法中，首先判断房子的剩余面积是否足够容纳这件家具，如果足够，则将家具对象添加到家具列表中，同时更新房子的剩余面积。如果不够，打印提示信息。
4. 创建一个家具类（Furniture），包括家具的名称和占地面积属性。
5. 通过创建多个家具对象，设置它们的名称和占地面积。
6. 创建一个房子对象（my_house），传入初始的户型和总面积。
7. 调用房子对象的添加家具方法，依次将家具对象添加到房子中。
8. 最后调用房子对象的打印信息方法（display）来展示房子的户型、总面积、剩余面积和家具名称列表。

完整代码

```
# 定义房子类
class House:
def __init__(self, house_type, total_area):
self.house_type = house_type # 户型
self.total_area = total_area # 总面积
self.furniture = [] # 家具列表

# 添加家具
def add_furniture(self, furniture):
if self.total_area >= furniture.area: # 判断剩余面积是否足够
self.furniture.append(furniture) # 添加家具到家具列表
self.total_area -= furniture.area # 更新剩余面积
print(f"{furniture.name}已添加到房子中")
else:
print(f"房子剩余面积不足，无法添加{furniture.name}")

# 展示房子信息
def display(self):
# 打印户型
print("户型:", self.house_type)
# 打印总面积
print("总面积:", self.total_area, "平米")
# 打印剩余面积
print("剩余面积:", self.total_area, "平米")
# 打印家具名称列表
print("家具名称列表:")
for furniture in self.furniture:#循环遍历家具列表，打印出每个家具的名字
print(furniture.name)
```

```
# 定义家具类
class Furniture:
def __init__(self, name, area):
self.name = name # 家具名称
self.area = area # 家具占地面积


# 创建房子对象
my_house = House("两室一厅", 100)

# 创建家具对象
bed = Furniture("床", 4)
wardrobe = Furniture("衣柜", 2)
table = Furniture("餐桌", 1.5)

# 添加家具到房子中
my_house.add_furniture(bed)
my_house.add_furniture(wardrobe)
my_house.add_furniture(table)

# 打印房子信息
my_house.display()
```

代码讲解

1. 定义了一个房子类（House），包含了房子的户型、总面积和家具列表属性。
2. 定义了一个家具类（Furniture），包含了家具的名称和占地面积属性。
3. 在房子类中，定义了添加家具的方法（add_furniture），用于判断剩余面积是否足够添加家具，并更新房子的剩余面积和家具列表。
4. 在房子类中，定义了展示房子信息的方法（display），打印房子的户型、总面积、剩余面积和家具名称列表。
5. 创建了一个房子对象（my_house）以及几个家具对象（bed、wardrobe、table）。
6. 调用房子对象的添加家具方法，将家具对象添加到房子中。
7. 最后调用房子对象的展示信息的方法，打印出房子的信息。

【练习】矩形面积和周长

项目简介

矩形面积和周长

知识模块

- Python 编程语言

知识点

- 静态方法
- 函数返回值与参数处理

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，创建一个几何图形计算程序，使用静态方法来计算矩形的面积和周长。

解题思路

1. 创建一个几何图形计算类
2. 使用 `@staticmethod` 装饰器定义静态方法

完整代码

```
class Geometry:
    @staticmethod
    def rectangle_area(width, height):
        return width * height

    @staticmethod
    def rectangle_P(width, height):
        return 2 * (width + height)


# 主程序
width = 4
height = 6
print(f"矩形的面积：{Geometry.rectangle_area(width, height)}")
print(f"矩形的周长：{Geometry.rectangle_P(width, height)}")
```

代码讲解

1. 静态方法 `rectangle_area` ： `@staticmethod` 装饰器：这个装饰器用于将下面的 `rectangle_area` 方法定义为静态方法，即可以通过类名直接调用。
2. `def rectangle_area(width, height)` :这个静态方法用于计算矩形的面积。它接收 `width` （宽度）和 `height` （高度）作为参数，然后返回宽度乘以高度的结果，即矩形的面积。
3. `def rectangle_P(width, height)` :用于计算矩形的周长。
4. 调用静态方法：使用类名 `Geometry` 和点号 `.` 来调用静态方法。
5. `Geometry.rectangle_area(width, height)` ：调用静态方法 `rectangle_area` 来计算矩形的面积。

【练习】面向对象跑步减肥

项目简介

- 跑步减肥

知识模块

- Python 编程语言

知识点

- 类、实例、构造方法、实例属性

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

小明和小美都爱跑步 小明体重 75 公斤 小美体重 45 公斤 每次跑步会减肥 0.5 公斤 每次吃东西体重增加 1 公斤 请根据打印出跑完步之后的体重

解题思路

- 使用面向对象方法进行操作

完整代码

```
class Person:
    # 构造方法
    def __init__(self, name, weight):
        # 两个实例属性
        self.name = name
        self.weight = weight

    # 打印实例对象会返回的内容
    def __str__(self):
        return f"名字：{self.name} 体重：{self.weight} 很爱跑步"

    # 跑步实例方法
    def run(self):
        print(f"{self.name} 在跑步，体重减少0.5公斤")
        self.weight -= 0.5

    # 吃饭实例方法
    def eat(self):
        print(f"{self.name} 在吃饭，体重增加1公斤")
        self.weight += 1


# 实例对象一：小明
xiaoming = Person("小明", 75)  # 实例化一个 Person 对象，名字为 "小明"，体重为 75
print(xiaoming)  # 打印 xiaoming 对象，会调用 __str__ 方法打印出对象的信息
xiaoming.eat()  # 调用 xiaoming 的 eat 方法，体重增加 1 公斤
xiaoming.run()  # 调用 xiaoming 的 run 方法，体重减少 0.5 公斤
print(xiaoming)  # 打印 xiaoming 对象，会调用 __str__ 方法打印出对象的信息

# 实例对象二：小美
xiaomei = Person("小美", 45)  # 实例化一个 Person 对象，名字为 "小美"，体重为 45
print(xiaomei)  # 打印 xiaomei 对象，会调用 __str__ 方法打印出对象的信息
xiaomei.eat()  # 调用 xiaomei 的 eat 方法，体重增加 1 公斤
xiaomei.run()  # 调用 xiaomei 的 run 方法，体重减少 0.5 公斤
print(xiaomei)  # 打印 xiaomei 对象，会调用 __str__ 方法打印出对象的信息
```

代码讲解

1. 首先，我们定义一个名为 `Person` 的类。在 `Person` 类中，我们定义了构造方法 `__init__`，用于初始化人的名字和体重。构造方法接受两个参数，分别是 `name` 和 `weight`，并将它们赋值给 `self.name` 和 `self.weight` 成员变量。
2. 接下来，我们定义了一个实例方法 `__str__`，用于返回一个字符串，表示人的名字、体重以及表示热爱跑步的信息。
3. 紧接着，我们定义了两个实例方法 `run` 和 `eat`，分别表示人在跑步和吃饭的行为。在 `run` 方法中，我们使用 `print` 函数输出跑步的信息，并通过 `self.weight -= 0.5` 将体重减少 0.5 公斤。在 `eat` 方法中，我们使用 `print` 函数输出吃饭的信息，并通过 `self.weight += 1` 将体重增加 1 公斤。
4. 最后，我们创建了两个 `Person` 类的实例对象 `xiaoming` 和 `xiaomei`，分别代表小明和小美。通过调用实例方法来模拟他们的行为，并使用 `print` 函数打印出相关信息。

【练习】青蛙跳井

项目简介

读写文件

知识模块

- Python 编程语言

知识点

- 类和对象
- 递归
- 封装

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序:青蛙跳台阶，共有10阶台阶，青蛙每次可以选择跳一阶或者两阶，

问：青蛙跳上这10个台阶共有多少种跳法。

解题思路

1. 我们可以发现，跳上第 n 个台阶的跳法数只与跳上前两个台阶的跳法数有关。如果我们知道了跳上第 n-1 个台阶和跳上第 n-2 个台阶的跳法数，就可以得到跳上第 n 个台阶的跳法数。
2. 特殊情况下，当只有 1 个台阶时，只有一种跳法；当有 2 个台阶时，有两种跳法。
3. 因此，我们可以使用递归的方式解决这个问题。递归的终止条件是当台阶数为 1 或 2 时，直接返回相应的值。
4. 在递归的过程中，通过递归调用 `jump` 方法计算跳上前两个台阶的跳法数，并将结果相加得到跳上第 n 个台阶的跳法数。
5. 为了避免重复计算，我们可以使用一个字典 `memo` 来记录已经计算过的结果，每次先检查字典中是否已经计算过当前台阶的跳法数。如果有，则直接返回对应的结果，否则进行计算并保存到字典中。 这样，通过递归调用和记录已计算结果的方法，我们可以得到青蛙跳上任意数量的台阶的跳法数。

完整代码

```
class Frog:
    def __init__(self):
        self.memo = {}                          # 用字典记录已经计算过的结果

    def jump(self, n):                          # 跳跃方法接受一个参数表示台阶数
        if n in self.memo:                      # 如果该台阶的跳法已经计算过
            return self.memo[n]                 # 直接返回计算结果
        if n == 1:                              # 如果只有一阶台阶
            return 1                            # 返回 1 种跳法
        if n == 2:                              # 如果有两阶台阶
            return 2                            # 返回 2 种跳法
        # 递归调用 jump 方法计算 n-1 和 n-2 台阶的跳法数
        res = self.jump(n-1) + self.jump(n-2)
        self.memo[n] = res                      # 将计算结果保存到字典中
        return res                              # 返回计算结果

frog = Frog()                                   # 创建 Frog 类的实例对象
num_of_ways = frog.jump(10)                     # 调用 jump 方法计算青蛙跳上 10 个台阶的跳法数
print(f"青蛙跳上这10个台阶共有{num_of_ways}种跳法。")    # 打印结果
```

代码讲解

1. 定义一个名为 `Frog` 的类，并在类的构造方法 `__init__` 中初始化一个空的字典 `memo`，用于记录已经计算过的结果。
2. 在跳跃方法 `jump` 中，首先检查字典 `memo` 中是否已经计算过台阶数 `n` 的跳法数。如果是，则直接返回已经计算过的结果。
3. 然后判断特殊情况，如果台阶数 `n` 为 1，则直接返回 1 种跳法；如果台阶数 `n` 为 2，则返回 2 种跳法。
4. 如果不是以上两种情况，递归调用 `jump` 方法计算 `n-1` 和 `n-2` 台阶的跳法数，并将结果相加得到 `n` 台阶的跳法数。
5. 将计算得到的结果保存到字典 `memo` 中，并返回该结果。
6. 创建一个 `Frog` 类的实例对象 `frog`，通过调用 `frog.jump(10)` 方法，计算青蛙跳上 10 个台阶的跳法数。
7. 使用 `print` 函数打印出结果。

【练习】随机密码

项目简介

随机密码

知识模块

- Python 编程语言

知识点

- 函数
- 字符串操作
- for循环
- random模块

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序，随机生成一个密码，密码包含大小写英文字母和数字的组合

解题思路

- 设置一个用于随机取出字符的基础字符串
- 循环 n 次，每次随机取出一个字符
- 各个字符拼接起来，保存到变量 result 中

完整代码

```
from random import choice  # 导入choice函数，用于从序列中随机选择一个元素

import string  # 导入string模块，用于获取ASCII字符

all_chs = string.ascii_letters + string.digits  # 大小写字母加数字，生成所有可选字符的字符串
#ascii_letters是string库中包含大小写英文字母的常量，digits是数字0-9的常量

#定义一个gen_pass
def gen_pass(n=8):#初始化是为8次

    result = ''  # 初始化一个空字符串，用于存储生成的密码

    for i in range(n):  # 循环n次，每次生成一个字符

        ch = choice(all_chs)  # 从all_chs中随机选择一个字符，赋值给ch
        #choice是random的一个函数，意思是从序列（可以使元组，列表，字符串等）中随机获取一个值

        result += ch  # 将选中的字符添加到结果字符串中

    return result  # 返回生成的密码字符串

if __name__ == '__main__':
    print(gen_pass())  # 调用gen_pass函数生成默认长度为8位的随机密码，并打印生成的密码

    print(gen_pass(4))  # 调用gen_pass函数生成长度为4位的随机密码，并打印生成的密码

    print(gen_pass(10))  # 调用gen_pass函数生成长度为10位的随机密码，并打印生成的密码
```

代码讲解

1. 首先，从 `random` 模块中导入了 `choice` 函数，该函数用于在序列中随机选择一个元素。同时，导入了 `string` 模块，用于获取 ASCII 字符集。
2. 创建了一个名为 `all_chs` 的变量，存储了大小写字母和数字的组合，即包含了所有可选的字符。
3. 定义了一个名为 `gen_pass` 的函数，该函数用于生成随机密码。函数接受一个参数 `n`，表示密码的长度，默认值为 8。
4. 在函数中，首先初始化一个空字符串 `result`，用于存储生成的密码。
5. 然后，通过循环 `for i in range(n):` 循环 `n` 次，每次生成一个随机字符。在循环中，使用 `choice(all_chs)` 从 `all_chs` 中随机选择一个字符，并将其添加到 `result` 字符串中。
6. 循环结束后，函数返回生成的密码字符串。
7. 在 `if __name__ == '__main__':` 中，进行了测试和输出：分别调用 `gen_pass()` 默认生成长度为 8 位

【练习】英雄游戏

英雄游戏

简介

实战思路

类与实例

- 相关知识点：类的定义、 属性、 方法
- 需求： 1. 定义一个英雄类，此英雄类需要包含 姓名、 血量、 攻击力、 还需要有一个方法为讲台词。

```
# 定义类
class Hero:
    # 定义类属性
    name = ""
    hp = 0
    power = 0

    # 定义方法
    def speak(self):
        print(f"欢迎来到英雄联盟, 我的名字是{self.name}")
# 类的实例化
hero = Hero()
# 实例对象.类属性 => 即可获取类属性
print(hero.name)
# 实例对象.方法名 => 即可调用实例
hero.speak()
```

- 相关知识点：构造函数、实例对象、实例属性、实例方法
- 需求： 1. 根据英雄类，实例化不同的英雄对象。 2. 每个英雄需要在实例化的时候，就有自己的姓名、攻击力、血量

```
class Hero:
    def __init__(self, name, hp, power):
        # 实例属性在构造函数内被初始化
        self.name = name
        self.hp = hp
        self.power = power

    # 定义方法
    def speak(self):
        print(f"欢迎来到英雄联盟, 我的名字是{self.name}, 我的血量为{self.hp}")


jinx = Hero("jinx", 1000, 100)
# 实例对象.方法名 => 即可调用实例
jinx.speak()
```

- 实例和类的关系总结： 1. 类只能获取、修改类变量。 1. 类不能直接调用实例方法，实例可以直接调用实例方法。 1. 实例对象可以获取类变量，但是当类变量和实例变量同名时，就只能获取到实例变量。 1. 实例不能修改类变量，可以修改获取实例变量。 1. self 就是实例本身。

封装、继承、多态

- 相关知识点：封装。
- 需求：每个英雄的血量不可以直接被获取或者修改。

```
class Hero:
    def __init__(self, name, hp, power):
        self.name = name
        # 私有属性
        self.__hp = hp
        self.power = power
jinx = Hero("jinx", 1000, 100)
# 报错
jinx.hp = 2000
```

- 相关知识点：继承、调用父类属性、父类方法、重写父类属性、方法、super。
- 需求1： 1. 现在除了英雄类，还有一种类是法师类。 1. 法师类继承于 Hero 类。 1. 法师类多了魔力的属性 1. 法师类多了一个放技能的方法。
- 需求2： 1. 定义战士类。 1. 战士类继承于Hero类。 1. 战士的会多一个护甲的属性。 1. 战士在初始化的时候需要多传入一个护甲信息。

```
class APCHero(Hero):
    def __init__(self, name, hp, power, mp):
        super().__init__(name, hp, power)
        self.mp = mp
```

```
    def speak(self):
        # print(f"欢迎来到英雄联盟，我的名字是{self.name}，我的血量为{self.hp}")
        super().speak()
        print("我是大美女")

    def charm(self):
        if self.mp < 50:
            print("蓝量不足")
        else:
            print("施展魅惑技能")
            self.mp -= 50


diaochan = APCHero("貂蝉", 1200, 80, 70)
diaochan.speak()
diaochan.charm()
```

- 相关知识点：多态、 类型注解、导包。
- 需求： 1. 定义一个单独的 fight 函数。 1. 在打斗之前，需要两个英雄先讲出台词。 1. 这个fight函数要求实现两个英雄的多轮回合制对打功能。最后需要返回一个赢家。 1. 创建一个测试用例文件，导入被测函数，并对它完成单元测试。

```
def fight(hero1: Hero, hero2: Hero):
    hero1.speak()
    hero2.speak()
    hero1_hp = hero1.hp
    hero2_hp = hero2.hp
    hero1_name = hero1.name
    hero2_name = hero2.name
    while True:
        hero1_hp = hero1_hp - 10
        hero2_hp = hero2_hp - 10
        # 当第一个英雄的血量小于0 或 当第二个英雄的血量小于0
        if hero1_hp <= 0 or hero2_hp <= 0:
            if hero1_hp > hero2_hp:
                # 字面量插值 - 字符串
                print(f"英雄{hero1_name}赢了")
                return hero1_name
            elif hero1_hp < hero2_hp:
                print("英雄赢了", hero2_name)
                return hero2_name
            else:
                return "平局"

def test_fight():
    jinx = Hero("jinx", 1000, 100)
    diaochan = APCHero("貂蝉", 1200, 80, 70)
    fight(jinx, diaochan)
```

设计模式

- 相关知识点： 不定长参数、工厂设计模式、 静态方法、类方法。
- 需求： 1. 现在多了一个同事小林，小林需要调用各种英雄初始化的方法，去完成他自己的逻辑。但是小林并不知道我设计了多少个类型的英雄。 1. 所以小林需要我将我目前所有的英雄类都放在一个工厂方法中进行初始化，传入不同的参数信息，返回不同的实例对象。如此一来，小林便不需要了解细节，只需要传入对应的参数获取对应的英雄即可。

```
class HeroFactory:
    @staticmethod
    def create_hero(hero_type, *args):
        if hero_type == "apc":
            return APCHero(*args)
        elif hero_type == "adc":
            # ADCHero(*args)
            print("adc")
        else:
            return Hero(*args)
```

三种方法对比

名称	定义	调用	关键字	使用场景
普通方法	至少需要一个参数self	实例名.方法名()	无	方法内部涉及到实例对象属性的操作
类方法	至少需要一个cls参数	类名.方法名() 或者实例名.方法名()	@classmethod	如果需要对类属性，即静态变量进行限制性操作
静态方法	无默认参数	类名.方法名() 或者实例名.方法名()	@staticmethod	无需类或实例参与

面试题

is和==的区别

在Python中，万物皆对象，而对象的三个基本要素:

- 内存地址
- 数据类型
- 值

而 `is` 与 `==` 都作为常用的判断语句去进行使用，这两者之间的主要区别是：

- `==` 运算符: 只比较两个对象的值，相同返回True，不同返回False。
- `is` 运算符: 比较两个对象的id，相同返回True，不同返回False。

在这种场景下， 两个判断的执行结果均为True。

```
a, b = 1,1
# 判断a, b 是否相等
print(a == b)
print(a is b)
```

```
a = [1,2,3]
b = [1,2,3]
# 对比值一致，返回True
print(a == b)
# 对比内存地址不一致，返回False
print(a is b)
```

Python深拷贝与浅拷贝

```
a = [1,2,3,[2,3,[4]]]
# 浅拷贝，此时内部嵌套的列表，b是直接引用的其内存地址。
b=a
# 此时修改内部嵌套列表对应的值
b[3][2][0]=1
# 会发现a内部嵌套的列表也被同步修改


# 深拷贝
b = deepcopy(a)
b[3][2][0] = 2
# 会发现b内部嵌套的列表不变
# ===
c = 1
d = 1
# True，不可变类型是值传递，所以共享的一个内存地址
id(c) == id(d)
# 不可变
c.copy() # 报错
```

Python元组和列表的区别

元祖和列表在Python中，都是有序的，可迭代的数据结构。

其中列表支持的操作较多，比如增删查改都是支持的。

元祖相较于列表本质的区别就在于元祖是**不可**变的数据结构，比如列表常用的增删改，在元祖这个数据结构中，原则上都是不可实现的。

- 修改、添加、删除操作

```
list_demo = [1, 2, 3]
tuple_demo = (1, 2, 3)
# 修改操作
list_demo[0] = "a" #成功
tuple_demo[0] = "a"  #报错
# 添加操作
list_demo.append("b") #成功
# 元祖没有可以添加的方法
# 删除操作
list_demo.remove(1) #成功
# 元祖没有可以删除的方法
```

- 访问、切片、遍历操作

```
list_demo = [1, 2, 3]
tuple_demo = (1, 2, 3)
# 访问
print(list_demo[0])
print(tuple_demo[0])
# 切片：获取0~1位的元素
print(list_demo[:2])
print(tuple_demo[:2])
# 遍历
for i in list_demo:
    print(i)
for i in tuple_demo:
    print(i)
```

- 占用内存

```
from sys import getsizeof
list_demo = [1, 2, 3]
tuple_demo = (1, 2, 3)
# 同样的元素，同样的数量，元祖小于列表
print(getsizeof(list_demo))
print(getsizeof(tuple_demo))
```

对比	元祖	列表
定义	(1, 2, 3)	[1, 2, 3]
修改	原则上不支持	支持
添加	不支持	支持
删除	不支持	支持
索引访问	支持	支持
切片	支持	支持
遍历	支持	支持
应用场景	固定的，不会被修改的数据	不固定的，可以被修改的数据
占用内存	较小	较大

【练习】员工薪资

项目简介

- 员工系统

知识模块

- Python 编程语言

知识点

- 实例方法、实例属性、类属性
- 构造方法、封装、继承、多态

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

设计一个简单员工系统设计，包括不同类型的员工（如全职员工和兼职员工）。每个员工有不同的薪资计算方法

解题思路

1. 创建一个Employee（员工）类，拥有一个抽象方法计算薪资的方法calculate_salary()。
2. 创建一个FullTimeEmployee（全职员工）类，继承自Employee类，实现calculate_salary()方法来计算全职员工的薪资。
3. 创建一个PartTimeEmployee（兼职员工）类，继承自Employee类，实现calculate_salary()方法来计算兼职员工的薪资。
4. 实例化一个FullTimeEmployee对象，设置工时为160，时薪为100，调用calculate_salary()方法计算并打印出全职员工的薪资。
5. 实例化一个PartTimeEmployee对象，设置工时为80，时薪为50，调用calculate_salary()方法计算并打印出兼职员工的薪资。 提示： - 可以使用abstractmethod装饰器定义抽象方法。 - 全职员工的薪资计算方法：薪资 = 工时 * 时薪 - 兼职员工的薪资计算方法：薪资 = 工时 * 时薪

完整代码

```
#创建员工类
class Employee:
    def __init__(self, hours, hourly_rate):
        self.hours = hours
        self.hourly_rate = hourly_rate

    def calculate_salary(self):
        pass


#创建全职员工类
class FullTimeEmployee(Employee):
    def __init__(self, hours, hourly_rate):
        super().__init__(hours, hourly_rate)  # 调用父类的构造方法来初始化属性

    def calculate_salary(self):
        return self.hours * self.hourly_rate  # 根据工时和时薪计算薪资

#创建兼职员工类
class PartTimeEmployee(Employee):
    def __init__(self, hours, hourly_rate):
        super().__init__(hours, hourly_rate)  # 调用父类的构造方法来初始化属性

    def calculate_salary(self):
        return self.hours * self.hourly_rate  # 根据工时和时薪计算薪资

full_time_employee = FullTimeEmployee(160, 100.5)  # 创建一个全职员工对象假设工作时长是160小时，薪资是100小时
full_time_salary = full_time_employee.calculate_salary()  # 调用全职员工对象的计算薪资方法
print("全职员工的薪资是:", full_time_salary)  # 打印全职员工的薪资

part_time_employee = PartTimeEmployee(80, 50)  # 创建一个兼职员工对象
part_time_salary = part_time_employee.calculate_salary()  # 调用兼职员工对象的计算薪资方法
print("兼职员工的薪资是:", part_time_salary)  # 打印兼职员工的薪资
```

代码讲解

1. 定义父类

```
class Employee:
    def __init__(self, hours, hourly_rate):
        self.hours = hours
        self.hourly_rate = hourly_rate

    def calculate_salary(self):
        pass
```

- 这部分定义了一个基类 `Employee`，其中包含了构造函数 `__init__`，该构造函数接收员工的的时长和薪资作为参数，并将它们存储为对象属性。还定义了一个未实现的 `calculate_salary` 方法，它在基类中只是一个占位符方法。

2. 定义子类

```
class FullTimeEmployee(Employee):
    def __init__(self, hours, hourly_rate):
        super().__init__(hours, hourly_rate)  # 调用父类的构造方法来初始化属性

    def calculate_salary(self):
        return self.hours * self.hourly_rate  # 根据工时和时薪计算薪资
```

- 这部分定义了 `FullTimeEmployee` 子类，它继承自基类 `Employee`。它的构造函数使用 `super().__init__` 来调用基类的构造函数，从而初始化继承的属性。`calculate_salary` 方法被重写，实现计算功能
- `PartTimeEmployee` 子类与 `FullTimeEmployee` 子类类似

3. 实例对象

```
full_time_employee = FullTimeEmployee(160, 100.5)  # 创建一个全职员工对象假设工作时长是160小时，薪资是100小时
full_time_salary = full_time_employee.calculate_salary()  # 调用全职员工对象的计算薪资方法
print("全职员工的薪资是:", full_time_salary)  # 打印全职员工的薪资

part_time_employee = PartTimeEmployee(80, 50)  # 创建一个兼职员工对象
part_time_salary = part_time_employee.calculate_salary()  # 调用兼职员工对象的计算薪资方法
print("兼职员工的薪资是:", part_time_salary)  # 打印兼职员工的薪资
```

- 实例化了不同类型的员工对象，即 `FullTimeEmployee` 和 `PartTimeEmployee`。然后，调用了它们的 `make_sound` 方法，由于多态性的存在，实际上调用了每个子类中不同的 `calculate_salary` 方法，分别输出不同的薪资

【练习】图书管理系统

项目简介

- 图书管理系统

知识模块

- Python 编程语言

知识点

- 类和对象
- 对象的属性和方法

受众

- 初级测试开发工程师
- 初级Python开发工程师

作业要求

编写一个Python程序: - 创建一个图书管理系统，实现以下功能： 1. 图书类 `Book` ：属性包括书名（ `title` ）、作者（ `author` ）和出版日期（ `publish_date` ），方法包括获取书名、获取作者和获取出版日期的方法。 2. 图书馆类 `Library` ：属性包括图书列表（ `books` ），方法包括添加图书、借出图书、归还图书和显示所有图书的方法。 3. `add_book` 方法：接受一个 `Book` 类型的参数，将其添加到图书列表中。 4. `borrow_book` 方法：接受一个字符串类型的参数（书名），找到对应书名的图书，并将其从图书列表中移除。 5. `return_book` 方法：接受一个 `Book` 类型的参数，将其添加到图书列表中。 6. `show_books` 方法：输出当前图书馆中所有图书的书名、作者和出版日期。

解题思路

1. 创建一个 `Book` 类，用于表示图书的信息，包括书名、作者和出版日期等属性。
2. 创建一个 `Library` 类，用于管理图书，包括添加图书、借出图书、归还图书和展示图书列表等方法。
3. 在 `Library` 类中，使用一个列表来存储图书列表，初始为空列表。
4. 在 `Library` 类的 `add_book` 方法中，将传入的图书对象添加到图书列表中。
5. 在 `Library` 类的 `borrow_book` 方法中，遍历图书列表，根据书名找到匹配的图书，并将其从图书列表中移除。如果找到了匹配的图书，则返回该图书对象；否则返回 `None` 。
6. 在 `Library` 类的 `return_book` 方法中，将传入的图书对象添加到图书列表中。
7. 在 `Library` 类的 `show_books` 方法中，遍历图书列表，逐个打印出图书的书名、作者和出版日期等信息。
8. 测试阶段，创建一个图书馆实例，并添加一些图书到图书馆中。
9. 调用图书馆的方法，如借出图书、归还图书和展示图书列表等，验证逻辑是否正确。

完整代码

```
class Book:
    def __init__(self, title, author, publish_date):
        self.title = title
        self.author = author
        self.publish_date = publish_date

    def get_title(self):
        return self.title

    def get_author(self):
        return self.author

    def get_publish_date(self):
        return self.publish_date


class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        """
    添加图书到图书馆的图书列表中
        """
        self.books.append(book)
```

```
    def borrow_book(self, title):
        """
    根据书名借出图书，如果找到对应书名的图书，则从图书列表中移除，并返回该图书对象
        """
        for book in self.books:
            if book.get_title() == title:
                self.books.remove(book)
                return book

    def return_book(self, book):
        """
     将归还的图书对象添加到图书列表中
        """
        self.books.append(book)

    def show_books(self):
        """
     显示当前图书馆中所有图书的书名、作者和出版日期
        """
        for book in self.books:
            print("书名：", book.get_title())
            print("作者：", book.get_author())
            print("出版日期：", book.get_publish_date())


# 测试
# 创建一个图书馆实例
library = Library()

# 创建几本图书并添加到图书馆中
book1 = Book("Python编程入门", "张三", "2021-01-01")
book2 = Book("Java编程基础", "李四", "2021-02-01")
book3 = Book("C++高级编程", "王五", "2021-03-01")

library.add_book(book1)
library.add_book(book2)
library.add_book(book3)

# 显示所有图书
library.show_books()

# 借出一本图书并打印出借出的书名
borrowed_book = library.borrow_book("Java编程基础")
if borrowed_book:
    print("借出的书：", borrowed_book.get_title())

# 再次显示所有图书
library.show_books()

# 归还图书并打印出归还的书名
library.return_book(borrowed_book)
print("归还书籍：", borrowed_book.get_title())

# 最后再次显示所有图书
library.show_books()
```

代码讲解

1. 创建图书馆实例 `library`。
2. 创建图书对象，并添加到图书馆中。
3. 调用 `show_books` 方法显示所有图书。
4. 调用 `borrow_book` 方法借出一本图书，并打印出借出的书名。
5. 再次调用 `show_books` 方法显示所有图书。
6. 调用 `return_book` 方法归还之前借出的图书，并打印出归还的书名。
7. 最后再次调用 `show_books` 方法显示所有图书。