目录

§0 前言 1
§1 Python 基本入门 1
§1.0 Python 基本介绍与安装 1
§1.0.0 Python 简介 1
§1.0.1 Python 安装 2
§1.0.2 Python IDE 5
§1.1 Python 基本语法 6
§1.1.0 注释 6
§1.1.1 字面量（常量、值） 7
§1.1.2 操作符 7
§1.1.3 变量 11
§1.1.4 输入与输出 12
§1.2 判断、循环、函数 13
§1.2.0 判断 13
§1.2.1 循环 17
§1.2.2 函数 19
§1.3 常用内置函数 21
§1.3.0 数据类型转化函数 21
§1.3.1 数学计算函数 24
§1.3.2 函数式编程工具 25
§1.3.3 系统交互函数 27
§1.3.4 其他函数 29
§1.4 内置数据类型 31
§1.4.0 数值类型(int, float, complex) 31
§1.4.1 布尔类型(bool) 32
§1.4.2 容器(list, tuple, set, dict, str) 32
§1.5 常用内置库与其中函数 35
§1.5.0 math模块 35
§1.5.1 random模块 36
§1.5.2 datetime模块 37
§1.5.3 os模块 40
§1.5.4 sys模块 42
§1.6 面向对象入门 45
§1.6.0 什么是面向对象？ 45
§1.6.1 类的基本结构 46
§1.6.2 理解self参数 47
§1.6.3 创建对象实例 47

§1.6.4 访问对象属性和方法 47
§1.6.5 类的内置属性 48
§1.6.6 对象的生命周期 48
§1.6.7 类的继承 49
§1.6.8 方法重写 49
§1.6.9 私有属性和方法 50
§1.7 Python 部分高级语法、次常用函数与语法糖 50
§1.7.0 二进制与位运算 50
§1.7.1 Python中的复合语句 52
§1.7.2 Python中的类型注释（Type Annotations） 56
§1.7.3 不定长参数（*args、**kwargs） 59
§1.7.4 try、except、finally 60
§1.7.5 for-else与while-else 62
§1.7.6 生成器（Generator）、迭代器（Iterator）与Range 63
§1.7.7 Python中类的魔法函数（magic method / special method） 63
§1.7.8 eval与exec 64
§1.7.9 断言（assert） 66
§1.7.10 匿名函数（lambda） 67
§1.8 图形交互初步 69
§1.8.0 Tkinter 简介 69
§1.8.1 Tkinter窗口搭建 69
§1.8.2 基本组件详解 70
§1.8.3 布局管理详解 80
§1.9 第三方库介绍及常用库推荐 82
§1.9.0 第三方库简介 82
§1.9.1 第三方库的安装与管理 82
§1.9.2 少数第三方库推荐 84
§2 简单算法介绍 86
§2.0 算法简介 86
§2.0.0 算法概述 86
§2.0.1 复杂度 86
§2.1 枚举算法 87
§2.1.0 简介 87
§2.1.1 复杂度 87
§2.1.2 要点 87
§2.2 模拟算法 88
§2.2.0 简介 88
§2.3 递归算法 88
§2.3.0 简介 88

§2.3.1 要点 88
§2.3.2 缺点 88
§2.4 分治算法 88
§2.4.0 简介 88
§2.4.1 要点 88
§2.5 贪心算法 89
§2.5.0 简介 89
§2.5.1 要点 89
§2.6 排序算法 89
§2.6.0 简介 89
§2.6.0 种类 89
§2.7 查找算法 97
§2.7.0 简介 97
§2.7.1 过程 98
§2.7.1 时间复杂度 98
§2.7.2 代码实现 98
§2.8 数据结构与算法 98
§2.8.0 数据结构的定义 98
§2.8.0 逻辑结构 99
§2.8.1 物理结构 100
§3 常见开发问题及解决方法 102
§3.0 Python常见报错及解决方法 102
§3.0.0 SyntaxError（语法错误） 102
§3.0.1 NameError（名称错误） 102
§3.0.2 TypeError（类型错误） 103
§3.0.3 IndexError（索引错误） 103
§3.0.4 KeyError（键错误） 104
§3.1 Debug工具简介及使用 104
§3.1.0 Pyhon Debugger（PDB） 104
§3.1.1 PyCharm Debugger使用 105
§3.1.2 VSCode Python Debugger 106
§3.2 其他常见Bug与排查方法 107
§3.2.0 逻辑错误 107
§3.2.1 无限循环 107
§3.2.2 模块导入错误 108
§3.2.3 文件路径问题 108
§3.2.4 内存问题 109
§3.3 通用调试技巧 109
§4 后言 109

§0 前言

因为当前本社的教材是基于C++语言进行创作的，而本社开发主要基于Python。兼之当前教材仍在编写并且完成度并不算高，且讲述的内容均为基本语法，有所欠缺，故特作此讲义。

此讲义初版由2024级社长编写，主要凭借社长个人知识及互联网查询进行编写，故此难有疏漏之处。之后的社长，如有需要，可在此版本上进行修订，但是希望可以在前言中保留以前版本的前言，并且在新版前言中增加修订内容。

此文档为讲义性质，而并非教材。其中内容大体按照教学顺序进行编排，但并不推荐按照此顺序进行教授。此教学讲义倾向于教学大纲、教学内容梗概、基本知识查询的手册。

§1 Python 基本入门

§1.0 Python 基本介绍与安装

§1.0.0 Python 简介

> Python is a programming language that lets you work more quickly and integrate your systems more effectively.
> ——Python.org

> Python is powerful... and fast;
> plays well with others;
> runs everywhere;
> is friendly & easy to learn;
> is Open.
> ——Python.org

> **The Zen of Python, by Tim Peters**
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea -- let's do more of those!

Python是现代编程开发中常用的高级语言，在各类软硬件均有广泛应用。正如上述引言所说，Python具有强大的能力，兼容性强，可移植性强，且易于学习，代码简洁易懂，环境开放（Tips: Python的运算速度并不快，其运算速度大约在C++的1/50至1/150之间），于是Python便逐渐成为了最受欢迎的编程语言之一。

Python是于1990年代初设计的，在至今的30余年间，Python经历了难以计数次的更新迭代。在此过程中，Python经历过三个大版本，分别为Python 1，Python 2，Python 3。其中Python 2于2000年10月16日发布，稳定版本是Python 2.7，于2020年1月1日起停止支持。虽然现在Python 2已经停止支持，现在网上部分的Python教程（尤指菜鸟编程）、示例代码仍在使用Python 2。所以当看到一些奇怪的Python代码时，可以怀疑这是Python 2。Python 2与Python 3并不完全相同，包括但不限于：

1. print函数不用加括号；
2. 许多包名不同，如Tkinter（Python 2)及tkinter（Python 3）；
3. 部分关键字、内置函数的语法修改或移除。

但总体上，Python 2与Python 3差距并不是特别大，故此并不需要去特别了解。

由于当下的Python通常特指Python 3，故此在本讲义的后续表述中，Python的大版本指Python 3.x中的x版本而不是Python 1、Python 2、Python 3；同样，Python的小版本指Python 3.x.y中的y版本。

现在Python仍在保持更新，进行性能优化以及功能拓展。截止初版讲义的此章节撰写时间（2024年12月15日），Python最新版本为Python 3.13.1，而学校机房电脑Python版本为3.7。请注意，在不同的Python版本之间，存在一定的语法差异和特性差异，且兼容此版本的第三方库也可能存在语法差异。故此，在对代码进行大版本迁移的过程中，请务必进行充分的测试。

§1.0.1 Python 安装

Python是支持多平台（Windows, Linux, MacOS）的语言，所以在安装过程中应当根据使用的平台进行选择。下面会对多平台的安装方法进行依次阐述。通常来说，本社社团场地的电脑上已经安装了Python，故此如果没有特殊需求，可以跳过此节。

Windows

方法一 Windows 商店

在电脑中没有安装Python时，在命令行（cmd或PowerShell或Win+R呼出的运行）中输入"python"或在Microsoft Store中搜索"Python"，即可打开/找到Microsoft Store中的Python，之后通过Microsoft Store下载安装即可。

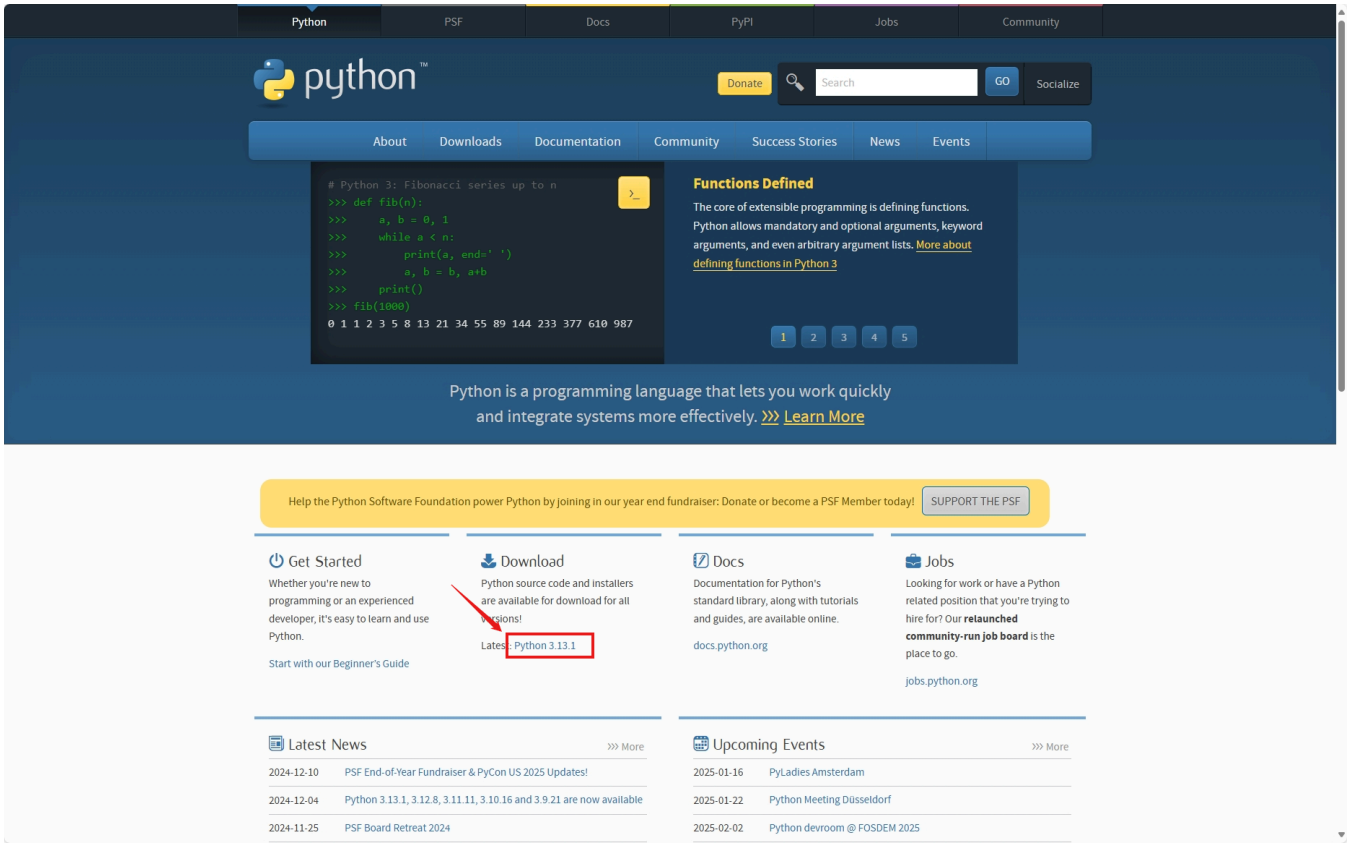
此方式的优点在于可以免去许多配置的麻烦，并且可以避免Python.org无法访问的情况。同时通过命令行进入商店时，可以自动跳转至最新版本。

但是需要注意的是，此方法仅支持Windows 10/11（其实也可能支持Windows 8，但是未曾测试，无法断言）。若直接在Microsoft Store中搜索Python，那么会出现许多不同的Python版本（据实测，会出现从3.7-3.13的所有大版本），令人眼花缭乱。

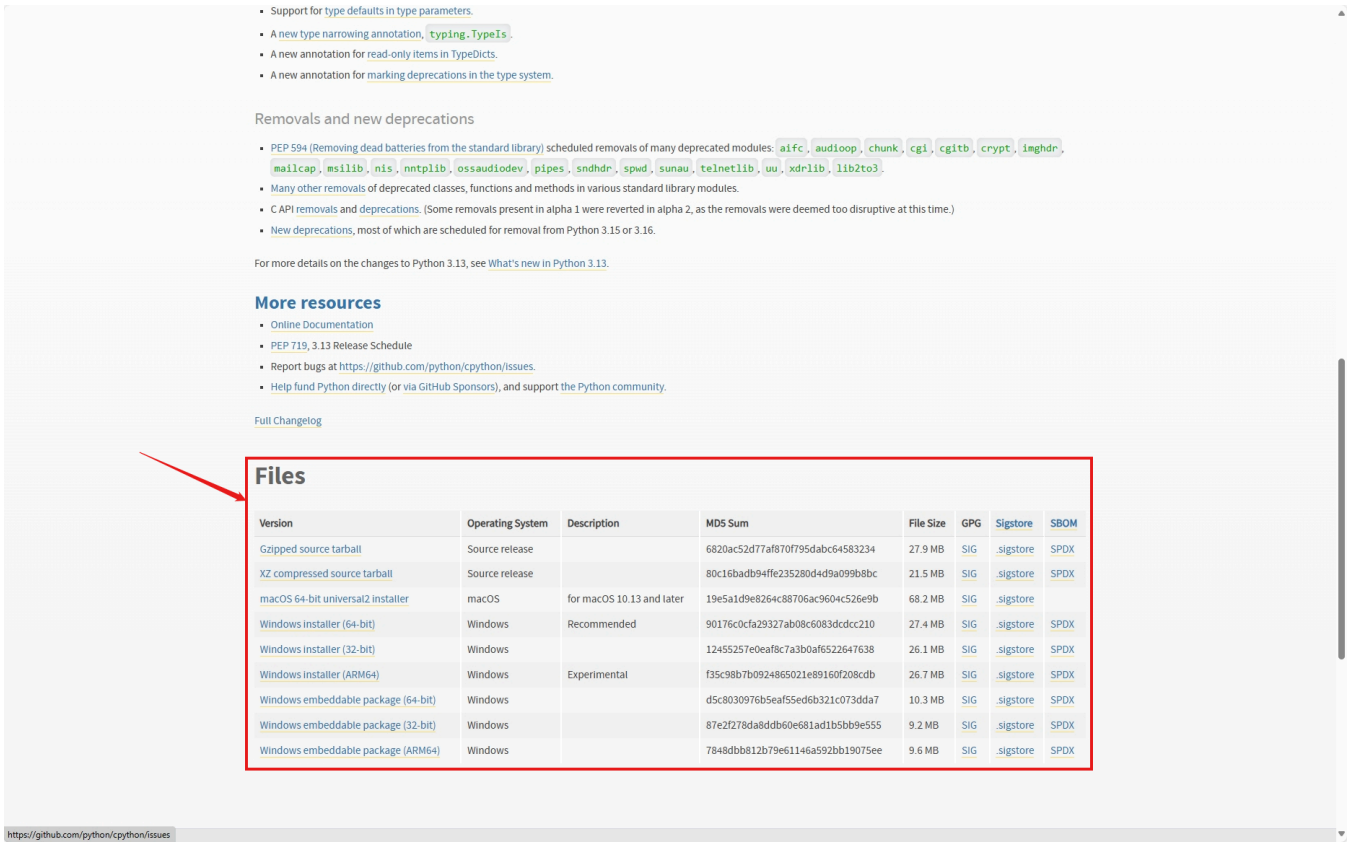
此方法缺点在于：1. 许多阉割版Windows中没有Microsoft Store或无法正常访问，且部分网络可能无法访问Microsoft Store，此时无法正常安装；2. 通过此方法安装Python过程中，无法自定义安装配置；3. 因为其安装与Python官网下载安装的Python有所差异，所以在使用过程中可能会有一些奇怪的Bug（在以前尤为常见，现在已经有了不少优化）。

方法二 官网下载

访问Python 官网 Python.org，在首页可以看到Download的标签即对应最新版Python的下载链接（如图1-1）。



进入后，滑动至页面最下方，找到"Files"及表格（如图1-2）。



通常来说，我们需要下载的版本为以下三种之一：

> Windows Installer (64-bit)
> Windows Installer (32-bit)
> Windows Installer (ARM64)

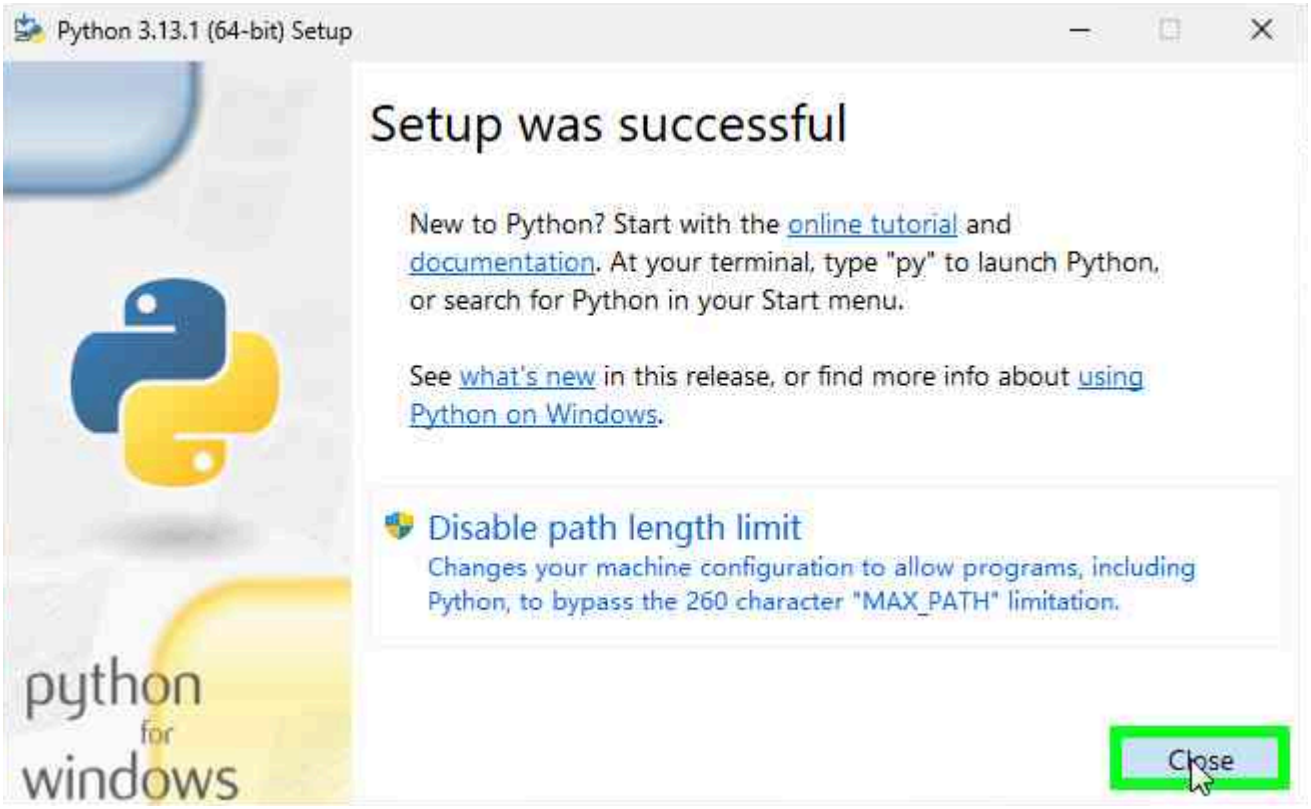
选择下载版本基本依据就是Windows的版本：若为32位Windows则下载Windows Installer (32-bit)；若为64位Windows则下载Windows Installer (64-bit)；若为Windows Arm则下载Windows Installer (ARM64)。关于Windows版本的判别此处不加阐述，如有困难可以在网上进行搜索。事实上，通常来说Windows Installer (64-bit)会是你所需要的。

确定下载版本后，点击表格中"Version"下对应版本文字（链接）即可下载。下载完成后运行即可进入安装器。

进入安装器后，可以看到安装界面（如图1-3），在安装前，推荐勾选"Add python.exe to PATH"（如框出部分），这样可以省去后续配置过程。若不勾选的后续配置过程略过。而关于"Use admin privileges when installing py.exe"，此选项会更改Python安装时需要的权限及安装的范围：若不勾选则仅为此用户安装，勾选则为系统所有用户安装。通常来说，此选项无需勾选。



之后，若需要自定义安装内容，可选择"Customize installation"进行自定义配置，通常来说无此需要，故略过，如有需要自行阅读安装器指引进行设置即可。点击"Install Now"即可进行一键安装。在出现"Setup was successful"（如图1-4）后即安装完毕，点击"close"即可。



特别的，关于"Disable path length limit"，此处不做解释。若需要了解相关内容，可以在B站上搜索/打开视频《30年前的技术债引发win11离奇bug，微软不敢修！》以进行了解。但总之，点了比不点好。

Linux

由于使用Linux的估计都是有开发基础的，故此Linux的安装过程略过。下仅指出几点注意点。

MacOS

首先，参照Windows方法二打开Python官网并且找到下载页（如图1-2）。新版Python仅支持64bit-MacOS，故此只有一个MacOS的下载选项，下载此文件即可。在老板Python下载中，请根据电脑版本进行下载，即根据电脑是32-bit或64-bit，具体判别方法请自行上网查找。但在下载时请注意其中的最低MacOS版本要求。

下载得到的文件为pkg类型文件，只需打开后一直按继续，同时在弹窗中点击同意即可。

由于本人并没有苹果设备，故不设图。

§1.0.2 Python IDE

实际上我们在上述过程中下载的是Python解释器，而通常来说如果我们不想用记事本写代码的话，我们还需要Python IDE来进行开发。如果你是在Windows或MacOS上的Python，并且按照§1.0.1中所描述的过程安装，那么Python的安装过程中会附带安装Python IDLE。这是一个简洁的IDE，并且无需额外配置，是一个轻量化且方便的选择。

不过，IDLE的设计比较复古，且功能单一，性能较弱，故此在此推荐一些好用的IDE。

Visual Studio Code

一个免费现代风格的轻量级IDE，支持语法高亮、代码补全，并且支持多平台。

优点：界面简洁，相对而言资源占用相对较少，插件丰富。对于各种开发语言、文件类型均有较好的支持。

缺点：各种语言的支持均通过插件实现，插件配置、环境配置相对麻烦。

PyCharm

JetBrains家族的专业Python IDE，功能多样，支持语法高亮、代码补全，并且支持多平台。

优点：专业软件，开发环境配置简单，页面优雅。功能众多，如可视化库管理、方便的Debugger等。

缺点：专业版为付费软件，免费版功能存在阉割（但实际上对我们社的一般开发没有影响）。资源占用高，在低配置电脑上运行可能出现问题。

Fleet

与 PyCharm 同属 JetBrains 家族的轻量化编辑器，对标 Visual Studio Code。虽然外观优雅且有一定优点，但实际上并不是特别推荐。目前该软件仍在预览阶段，故此建议以后修订本讲义时，修订者可以尝试一下，如果有更多的优化，建议及时修改。

优点：相较Visual Studio Code，配置简单，可以自动识别，页面简洁。

缺点：当下仍在 Public Preview（截至初稿），依赖于 JetBrains Toolbox 使用，每42天强制更新；代码补全较难用；插件市场仍有待发展。总之，未来可期。

Atom

一个基于Web技术构建的编辑器，启动速度快，支持多语言语法高亮，插件主题丰富，支持多平台。

优点：轻量化，原生Markdown支持，开源免费，原生Git支持。

缺点：其本质是一个文本编辑器，并不具备作运行代码框架的能力。

§1.1 Python 基本语法

Python是一个解释型语言。所以说，在程序运行前，并不会预检查代码语法的正确性。故此，编写过程中的语法的正确与否是一个重要的事情，否则就可能在开发的过程中埋坑。

> **Tip**
>
> 解释型语言：相对于编译型语言（如C++）存在的，源代码不是直接翻译成机器语言，而是先翻译成中间代码，再由解释器对中间代码进行解释运行。——百度百科

在本章节中仅对Python语法的基础部分进行讲解，即大纲式的讲解，目的在于描述、构建一个完备的、Python式的语言逻辑。故此，对于许多语法的具体讲解将会在后续章节中展开，而并不会在此章节中赘述。

若对于一些地方存在不理解，可以选择尝试在其他章节中寻找答案、利用互联网进行查找，或是在Python 官方文档寻找答案。

> **Important**
>
> 在本讲义后续的讲述中，如无特别说明，一切符号均指半角符号，即英文符号。

§1.1.0 注释

注释，即在代码中，用于注明信息给开发人员看，而不会被电脑识别的内容。通常来说，注释分为单行注释和多行注释。

单行注释，由`#`号引导。在每行代码中，`#`号后的内容均为注释内容而不会被运行，而`#`前的内容不会受影响。`#`可以位于代码行的最开头，也可以位于结尾。

多行注释，在注释前后均加上`"""`或`'''`（即三个双引号或三个单引号）。

> **Important**
>
> 此引号要求前后对应，即不能以`"""`开头而以`'''`结尾。在一对`"""`之间的`'''`或一对`'''`之间的`"""`并不会影响注释，并且不会中断注释。

事实上，所谓多行注释实际上就是多行字符串的字面量。具体参见本节的“字面量”章节及§1.3节中的`字符串章”节。但是由此可以知道：为了防止一些隐蔽的 Bug，我们应当减少多行注释的使用而多使用单行注释。

关于注释规范的问题，正如上文所说，通常来说注释是通过单行注释来写的，但是有一种情况例外，就是对类及函数写文档/帮助的时候。在这种时候，应当紧跟着类/函数的声明的下一行写多行注释以保证 Python 内置的`help`函数可以有效读取到类/函数的文档/帮助。这一段话初学时看不懂没有关系，将会在本节的“函数”章节以及§1.5中的“Python中的类”章节进行讲解。

§1.1.1 字面量（常量、值）

字面量，即在代码中，被写下来的固定值，如：

```
123 # int
12.13 # float
2 + 3j # complex
True # bool
'Hello World' # str
f'Hello {1}' # str
[1, 2, 3] # list
(1, 1, 4, 5) # tuple
{1: 2, 3: 4} # dict
{1, 2, 3, 4} # set
```

在后面注释代表了这些字面量的数据类型，对于初学者，在初次阅读此段时并不需要完全理解数据类型相关内容。关于数据类型会在后续的§1.3中进行详细介绍。不过所有的字面量类型均已在此列出。

> **Tip**
>
> 你知道吗？虽然list、tuple、dict、set如上写出，也是字面量的形式，但是在Python官方文档中，并没有将他们列在字面值之中。
> 除此之外，上示例中包括两个str，其中第二个str在Python官方文档中被称作fstr，在§1.4中的str节会有介绍。

§1.1.2 操作符

在Python中，操作符是用于执行对变量（在后文介绍）和值的操作。Python中的有多种操作符，下面将会依次介绍。

通过句法逻辑的性质进行划分，可以将操作符划分为运算符与分隔符。

通过表现形式进行划分，可以将运算符分为括两种：符号运算符和关键字运算符。简单来说，符号运算符是特殊字符或其组合，而关键字运算符则类似于英文单词。

符号运算符包括：

```
+       -       *       **      /       //      %      @
<<      >>      &       |       ^       ~       :=
<       >       <=      >=      ==      !=
```

关键字运算符种类较多，且由于官方文档没有明确划分，这里不统一列出。但是在下述具体讲解中会包括其中大部分常用的关键字运算符。

分隔符包括：

```
(       )       [       ]       {       }
,       :       !       .       ;       @       =
->      +=      -=      *=      /=      //=     %=
@=      &=      |=      ^=      >>=     <<=     **=
```

关于分隔符，在此不细说，大部分会在后续的教材中涉及。

由于对于相关知识的需求，此部分建议结合§1.4章节进行阅读。此段中默认对于各种数据类型有一定的了解。除此之外，相关介绍在对应篇目中也会进行更多详细介绍。

一元算术和位运算

一元算数运算符包括：正`+`、负`-`，这与数学中通常认为的正负号相同。

一元位运算符仅包括：按位取反`~`。关于位运算，此处略过，将在§1.7的相关内容进行介绍。只需知道，除非额外定义，在Python中，`~`只对int类型进行了定义，并且`~x`被定义为`-(x+1)`。

二元算术运算符

二元算术运算符包括：加`+`、减`-`、乘`*`、乘方`**`、除`/`、整除`//`、取模`%`、at`@`，除`@`通常用在数字运算上。

首先，这些运算符均可用在两个int、int与float、两个float之间，但效果并不完全相同。对于两个int之间的运算，返回结果仍为int（除`/`及`**`后接负数的情况以外），而任意一者为float，则返回结果为float。特别的，/无论是连接int还是float，即便结果是整数，返回的也是float类型。

`+`、`-`、`*`、`/`与正常数学中的$+$、$-$、$\times$、$\div$基本一致，但是由于浮点数的缺陷，可能会导致一些误差（在§1.4中的float章节会简单讲述）。

对于`**`，即乘方，与许多数学表述中的^（但是`^`在Python中并不被如此定义）相近。例如，2^2在Python中即可表示为`2 ** 2`。对于开根，在Python中也可以通过将根号转变为分式次方来表示，例如$\sqrt{2}$在Python中即可表示为`2 ** 0.5`（其实`2 ** 0.5`并不代表$\sqrt{2}$，而是其近似值。如果需要表述精准的$\sqrt{2}$等数，可以使用一些科学计数的库，如numpy）。

> **Tip**
>
> 虽然正常数学中并没有对0^0做出定义，但是在Python中，`0 ** 0`的运算结果为1。
>
> 事实上，这是合理的，因为：
>
> $$\lim_{x\to 0} x^0 = 1$$

> **Tip**
>
> 你知道吗？在Python官方文档中，并没有将`**`归类为二元算数运算符，而是单独拎出，称作幂运算符。

对于`//`，即整除，基本等价于/后再进行下取整，比如：

```
1 // 2 == 0
5 // 3 == 1
1.5 // 2 == 0.0
2 // 0.5 == 4.0
```

但是由于浮点数的误差，有的时候在对float进行`//`的时候，可能会出现错误，如`9.6//0.8`的结果是11.0而非理论上的12.0。其原因在§1.4中的float章节会简单讲述。

注意：不论是`/`还是`//`，若尝试除以0（或0.0），或是尝试对0（或0.0）作负幂次的`**`运算，则会导致`ZeroDivisionError`异常。

对于%，即取模，`a % b`等价于`a - (a // b) * b`。

对于`@`，其目标是用于矩阵乘法。但是到目前为止，并没有内置的 Python 类型实现此运算符。

二元位运算符

二元算术运算符包括：左位移`<<`、右位移`>>`、位与（AND）`&`、位或（OR）`|`、位异或（XOR）`^`。与`~`一样，除非额外定义，位运算符均只对int类型进行了定义。此处不作更多介绍，详见§1.7的相关内容。

比较运算符

> ⚠ **Warning**
>
> 请注意，在Python中，比较数值是否相等，所使用的等号应当是`==`而不是数学中、日常生活中的`=`。而在Python中，`=`表示的意思是将等号右值赋值给左量，详细参见§1.1.3中的相关内容。

Python中的比较运算符包括：

```
<       >       <=      >=      ==      !=      is
```

Python中的大部分的比较运算符都是顾名思义的，故此不再特殊进行介绍。只需要知道`!=`代表的是不等于即可。

当然，`is`虽然含义浅显，但是应该与`==`进行区别：简单来说，`==`是比较内容是否相等，而`is`是比较其在Python中是否指向同一对象，更详细的详见§1.7。

逻辑运算符

```
and     not     or
```

这三个是用于对若干个布尔值（`True`、`False`）进行运算，同样也是顾名思义。简单来说，遵循以下规则：

- `and`：二元运算符，当且仅当两边连接的两个值均为`True`时结果才为`True`，否则为`False`。
- `or`：二元运算符，当且仅当两边连接的两个值均为`False`时结果才为`False`，否则为`True`，换言之只要有一个值为`True`则返回`True`。
- `not`：一元运算符，输出与传入值相反的结果，即输入`True`，输出`False`，反之亦然。

具体使用示例如下：

```
1 > 2 and 2 > 1     # 结果: False
1 > 2 or 2 > 1      # 结果: True
not 1 > 2           # 结果: True
not 2 > 1           # 结果: False
```

> **Tip**
>
> 对非bool类型的对象进行逻辑运算符运算时，运行逻辑如下：
>
> | 逻辑运算符 | 运算结果 |
> | --- | --- |
> | and | 返回第一个为假的值，如果所有值都为真，则返回最后一个值。 |
> | not | 始终返回布尔值（`True`或`False`），根据对象的真值取反。 |
> | or | 返回第一个为真的值，如果所有值都为假，则返回最后一个值。 |
>
> 其中对于非bool类型对象的真假判断，遵循以下规则：
>
> - 数值类型：0为`False`，非0为`True`
> - 容器类型：空容器为`False`，非空为`True`
> - `None`始终为`False`

在日常使用中，你应当要注意逻辑运算符的两个重要点：

1. 逻辑运算符的运算先后顺序

   这和平时的`+ - * ** / // %`等的运算顺序类似，其运算先后满足：先`not`，再`and`，最后`or`，平级之间从左到右依次运行，但是这些运算均在`+ - * ** / // %`以及`== != > < >= <=`等之外（在没有小括号的情况下。如有小括号，则先括号内再括号外，具体则与`+ - * ** / // %`类似）。具体为：

运算符	优先级	示例等价写法
`not`	最高	`not x and y` → `(not x) and y`
`and`	中等	`x and y or z` → `(x and y) or z`
`or`	最低	`x or y and z` → `x or (y and z)`

   > **Tip**
   >
   > 建议无论如何，都按照所期望的运行方式添加小括号。

   例子如下：

```
print(True or False and False)  # 输出：True
print(not False and True)  # 输出：True
# 小括号改变运算顺序
print((True or False) and False)  # 输出：False

# 实际运用实例
# 检查用户名是否既不是空，也不是"admin"
username = ""
if not username or username == "admin":
    print("无效用户名")
# 等价于：
if (not username) or (username == "admin"):
    print("无效用户名")
```

2. 逻辑运算符运算的截断特性

   两个二元逻辑运算符`and`、`or`均具有截断特性（即短路求值），也就是它们在计算时会根据前面的表达式结果决定是否继续执行后面的表达式，具体如下（以下从左到右依次计算指连续的`and`和`or`，连续指这些`and`或`or`之间不隔有更低级的运算符）：

   - `and`截断特性：从左到右依次计算，如果遇到一个`False`值，立即返回该值，不再继续计算，例：

     ```
     print(3 and 5)       # 输出：5（3为真，继续计算5，返回最后一个值）
     print(0 and 10)      # 输出：0（0为假，直接返回，不再计算10）
     print([] and "Hi")   # 输出：[]（空列表为假，直接返回）
     ```

   - `or`截断特性：从左到右依次计算，如果遇到一个`True`值，立即返回该值，不再继续计算，例：

     ```
     print(3 or 5)        # 输出：3（3为真，直接返回，不再计算5）
     print(0 or "Hi")     # 输出："Hi"（0为假，继续计算"Hi"）
     print([] or 0)       # 输出：0（[]和0都为假，返回最后一个0）
     ```

§1.1.3 变量

相较于定值的字面量，变量是在开发过程中更常使用的。变量的用处主要为能够储存计算结果或者表示值的抽象概念（此处通常是对应类型的实例即object，相关内容会在§1.6中进行讲解）。

每一个变量都有自己的名称，用于标识。在使用过程中，只需直接写出此名称即相当于调用此变量。在代码编写过程中，在定义后通常不能修改变量的名称。相关实例将会在后文中一并展现。

变量在使用前需要定义，在使用过程中常常需要赋值。故此下面介绍变量的定义、赋值格式。在Python中，变量没有专门的定义式子，而是直接将定义融合在了赋值的过程中。Python的定义、赋值格式如下：

```
变量名称 = 变量的值
```

在此赋值式子中，若在之前变量名称没有被用于标识某个变量，即没有名为此的变量，则代表这里是定义这个变量并且给其赋值为等号右边的值。而如果已经存在名为此的变量，那么并不会重新创建一个变量，而是直接将原变量修改为等号后的变量的值。

当然，除了使用`=`进行赋值之外，在Python3.8+中，你也可以使用海象运算符（`:=`）进行赋值。海象运算符的用处在于在赋值的同时，将此值进行返回。其逻辑类似于C++等语言中的赋值。例如：

```
a = 1
print(a=2)      # 报错: TypeError: 'a' is an invalid keyword argument for print()
print((a=2))    # 报错: SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of
'='?
print(a:=2)     # 输出: 2
print(a)        # 输出: 2
```

> ⚠ **Warning**
>
> 海象运算符只能用于赋值，而不能用于声明变量，如：
>
> ```
> # (a未声明)
> print(a:=1)     # 报错: SyntaxError: invalid syntax
> a = 1
> print(a:=2)     # 输出: 2
> ```
>
> 在使用海象运算符时，请注意勤打括号，否则很容易出现语法问题/逻辑问题。

§1.1.4 输入与输出

此处介绍Python中最基本的输入与输出方法，即print与input函数。其实本段本应放在§1.3章进行介绍，但是为了保证一个完整的基础Python基本框架，所以在此处对输入输出进行简要介绍。一些详细的介绍仍在§1.3章节中。

print

print为Python的基本输出函数，其基本使用方式如下：

```
print(...)
```

其中...即你所想要输出的对象，可以是字面量，也可以是变量，也可以是运算的式子，甚至可以是一些奇怪的东西，但是具体可以输出什么、输出是什么样的，这与Python作为面向对象语言有关，参见§1.6相关内容。

通常来说，一个print可以仅输出一个对象，也可以同时输出多个对象。输出多个对象时，应使用逗号分隔每一个欲输出的对象。若无特别传参（即sep参数，详见§1.3的print节），则多个对象之间使用单个空格分隔（注意：在最后一个对象输出后并不会有一个空格）。无论是输出一个对象还是多个对象，在所有对象之后，若无特殊传参（即end参数，详见§1.3的print节），则会再输出一个“\n”即换行符，俗称回车（“\n”是一个转义符，详见§1.4的str节；同时,）。

下面给出一些例子。

```
代码: print('我讨厌六楼的教室')
输出: 我讨厌六楼的教室
```

```
代码: print(1 + 1 - 2 ** 9)
输出: -510
```

```
代码: print(1, 2, 3, 4, 5)
输出: 1 2 3 4 5
```

```
代码: print([1, 2, 3, 4, 5], {1: 3, 4: 5, 2: 7})
输出: [1, 2, 3, 4, 5] {1: 3, 4: 5, 2: 7}
```

```
代码:
print('欢迎来玩')
print('CHUNITHM !!!')
输出:
欢迎来玩
CHUNITHM !!!
```

input

input为Python的基本输入函数，其基本使用方式如下：

```
input(...) 或 input()
```

input可以接受文本输入。在使用该函数时，代码不会继续运行直到完成输入接收。input函数会一直接收输入直到收到换行符（即回车）。在接收到换行符之后，input函数会将input函数开始后直到换行符之间的全部字符（此处各平台、各IDE、各CLI驱动软件间存在一定差异，在§1.3的input节会作少许讲述。）作为字符串作为函数返回值传回代码。无论输入的是什么，input均会以str类型返回输入值。

虽然input函数以换行符为接收输入的截止符，但是返回的str中，并不会包括末尾的“\n”即换行符。

关于input的参数，可以没有，也可以有。对于存在参数的input函数，其等价于print函数+无参数的input函数，即：

```
input(obj)
等价于如下的函数f:
def f(obj):
    print(obj, end='')
    return input()
```

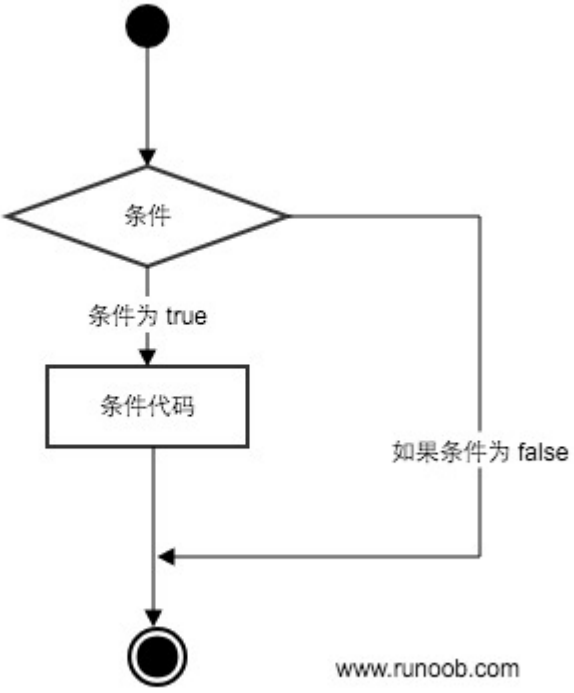
请注意：obj并不仅限于str，但是并不如print一般支持多个参数。

§1.2 判断、循环、函数

§1.2.0 判断

在Python中，条件判断（也称为分支结构）允许程序根据特定条件执行不同的代码块，类似于现实生活中的决策过程。

Python通过语句的执行结果（`True`或`False`）来决定执行哪个代码块。以下是条件判断的基本示意图：



if语句

Python中if语句的基本语法如下：

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

执行逻辑：

1. 当"condition_1"为`True`时，执行"statement_block_1"
2. 当"condition_1"为`False`时，检查"condition_2"
3. 当"condition_2"为`True`时，执行"statement_block_2"
4. 当"condition_2"为`False`时，执行"statement_block_3"

> **Important**
>
> - 每个条件后必须使用冒号`:`
> - 使用缩进（Tab或空格）划分语句块
> - `elif`是`else if`的简写
> - 非布尔值会隐式转换为布尔值，其规律已在§1.1.2中逻辑运算符中进行讲述

示例代码：

```
var1 = 100
if var1:
    print("1 - if表达式条件为True")
    print(var1)

var2 = 0
if var2:
    print("2 - if表达式条件为True")
    print(var2)
print("Good bye!")
```

输出结果：

```
1 - if表达式条件为True
100
Good bye!
```

常用比较运算符：

运算符	描述
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于（注意不是单等号）
!=	不等于

数字猜谜游戏示例：

```
number = 7
guess = -1
print("数字猜谜游戏!")
while guess != number:
    guess = int(input("请输入你猜的数字："))

    if guess == number:
        print("恭喜，你猜对了！")
    elif guess < number:
        print("猜的数字小了...")
    elif guess > number:
        print("猜的数字大了...")
```

if嵌套

可以在if语句中嵌套其他if语句：

```
if condition_1:
    if condition_2:
        ...
    elif condition_3:
        ...
    else:
        ...
elif condition_4:
    ...
else:
    ...
```

示例：

```
num = int(input("输入一个数字："))
if num % 2 == 0:
    if num % 3 == 0:
        print("你输入的数字可以整除2和3")
    else:
        print("你输入的数字可以整除2，但不能整除3")
else:
    if num % 3 == 0:
        print("你输入的数字可以整除3，但不能整除2")
    else:
        print("你输入的数字不能整除2和3")
```

match...case语句（Python 3.10+）

`match...case`提供了更简洁的条件判断方式：

```
match subject:
    case pattern_1:
        action_1
    case pattern_2:
        action_2
    case pattern_3:
        action_3
    case _:
        default_action
```

HTTP状态码示例：

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"

print(http_error(400))  # 输出: Bad request
```

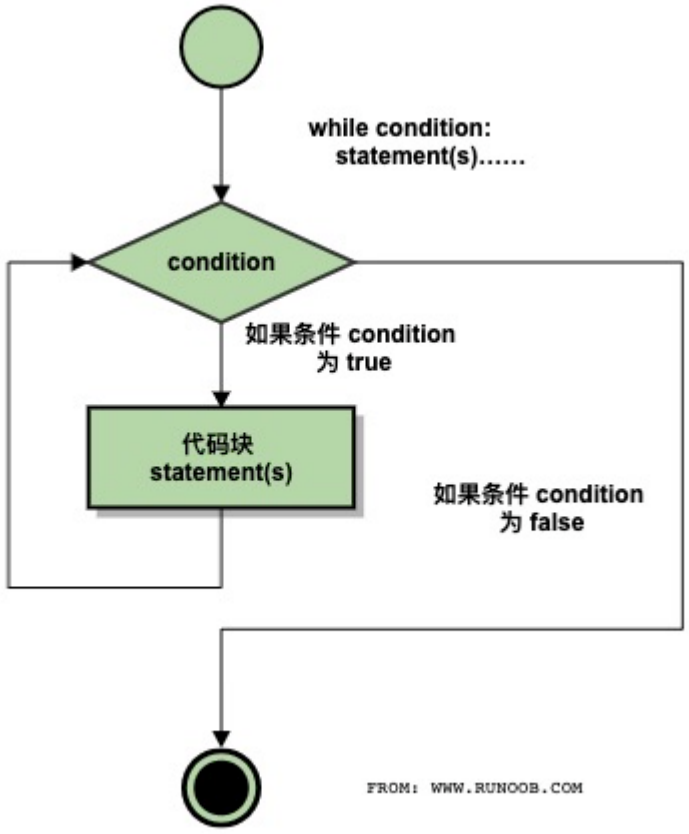
§1.2.1 循环

Python提供了两种循环结构：

- `for`循环：用于遍历序列中的元素
- `while`循环：基于条件判断的重复执行

while循环

流程图如下：



基本语法：

```
while condition:
    statements
```

其运行过程为：判断`condition`，如为`True`则运行`statements`代码块，运行结束后再次进行此过程，否则结束循环。

例，计算1到100的和：

```
n = 100
sum = 0
counter = 1
while counter <= n:
    sum += counter
    counter += 1
print(f"1到{n}之和为: {sum}")
```

无限循环示例：

```
while True:
    num = int(input("输入一个数字: "))
    print("你输入的数字是:", num)
```

> **Tip**
> 终止无限循环的方法：
>
> - `Ctrl+C`（抛出`KeyboardInterrupt`）
> - 任务管理器结束进程
> - `Ctrl+D`（在命令行中）

for循环

基本语法：

```
for variable in sequence:
    statements
```

遍历列表示例：

```
games = ["O.N.G.E.K.I", "CHUNITHM", "MaiMai DX"]
for game in games:
    print(game)
```

遍历字符串示例：

```
word = '玛卡巴卡'
for letter in word:
    print(letter)
```

数列递推（较为常用）：

```
for i in range(a[, b[, c]]):    # c != 0
    ...
```

其可大致等价为`C`/`Java`的：

```
for(int i = 0; i < a; i++) { ... }  // 仅存在参数a
for(int i = a; i < b; i++) { ... }  // 仅存在参数a, b
for(int i = a; i < b; i+=c) { ... } // 存在参数a, b, c且c>0
for(int i = a; i > b; i+=c) { ... } // 存在参数a, b, c且c<0
```

而其中单变量的`range`更为常用。如果看不懂以上`C`/`Java`代码，则你可以直接理解：对于

```
for i in range(a):
    ...
```

便相当于对于`for`后的代码块循环a次，而对于第$k(k=1,2,\dots,a)$次循环中，循环内的i取值便为$k-1$，即i的取值为从0到$a-1$。

关于`range`函数，详见§1.3.4相关内容。

循环控制语句

- `break`：终止整个循环
- `continue`：跳过当前迭代
- `pass`：空操作，保持结构完整，事实上可以直接使用`...`（三个英文句号点）代替

示例：

```
# break示例
for letter in 'Runoob':
    if letter == 'b':
        break
    print(letter)

# continue示例
for letter in 'Runoob':
    if letter == 'o':
        continue
    print(letter)

# pass示例
for letter in 'Runoob':
    if letter == 'o':
        pass
        print('执行pass块')
    print(letter)
```

§1.2.2 函数

函数是可重用的代码块，用于实现特定功能。

定义函数

基本语法：

```
def function_name(parameters):
    """文档字符串"""
    function_body
    return expression
```

示例：

```
def max(a, b):
    return a if a > b else b

print(max(4, 5))  # 输出: 5
```

参数传递

Python参数传递特点：

- 不可变对象：类似值传递（如整数、字符串）
- 可变对象：类似引用传递（如列表、字典）

示例：

```
def change_list(mylist):
    mylist.append([1,2,3,4])

mylist = [10,20,30]
change_list(mylist)
print(mylist)  # 输出: [10, 20, 30, [1, 2, 3, 4]]
```

直接来说，对于不可变对象，在函数内对于不可变对象的参数进行的修改，在函数结束时不会传回；但是对于可变对象，在函数内对其进行修改后，在函数结束时其修改则会保留（参考上示例）。

参数类型

1. 必需参数
2. 关键字参数
3. 默认参数
4. 不定长参数（参见§1.7.3）

示例：

```python
# 默认参数
def printinfo(name, age=35):
    print(f"名字: {name}, 年龄: {age}")

# 不定长参数
def printinfo(arg1, *vartuple):
    print("输出:")
    print(arg1)
    print(vartuple)
```

匿名函数

使用`lambda`创建匿名函数，详见§1.7.10：

```python
sum = lambda a, b: a + b
print(sum(10, 20))  # 输出: 30
```

return语句

用于从函数返回结果：

```python
def sum(a, b):
    total = a + b
    return total

result = sum(10, 20)
print(result)  # 输出: 30
```

§1.3 常用内置函数

§1.3.0 数据类型转化函数

以下函数用于各种数据类型的转化，建议先阅读§1.4后再深入了解此章节。

事实上，在了解Python的面向对象相关语法后，你会知道，以下“函数”实为创建类的对象，调用对应类的`__init__`函数。

int

功能：将字符串或数字转换为整数

示例：

```
# 基本转换
print(int('123'))      # 123
print(int(12.34))      # 12

# 指定进制
print(int('1010', 2))  # 10 (二进制转十进制)
print(int('A', 16))    # 10 (十六进制转十进制)

# 错误处理
try:
    print(int('12.3'))  # ValueError
except ValueError as e:
    print(f"错误：{e}")
```

常见问题：

1. 转换浮点数字符串会报错
2. 无效字符会导致`ValueError`

注意事项：

- 字符串必须符合整数格式
- 浮点数转换会直接截断小数部分
- 指定进制时，字符串必须符合该进制表示法

float

功能：将字符串或整数转换为浮点数

示例：

```
print(float('123'))     # 123.0
print(float('12.34'))   # 12.34
print(float(10))        # 10.0

# 科学计数法
print(float('1e-3'))    # 0.001

# 错误处理
try:
    print(float('abc'))  # ValueError
except ValueError as e:
    print(f"错误：{e}")
```

注意事项：

- 支持科学计数法表示
- `NaN`和`Infinity`会被特殊处理
- 精度问题：`0.1 + 0.2 != 0.3`（浮点数精度问题，二进制必然会导致的结果。对于十进制小数的高精表示，可自行查阅相关资料）

str

功能：将对象转换为字符串表示

示例：

```
print(str(123))        # '123'
print(str(12.34))      # '12.34'
print(str([1,2,3]))    # '[1, 2, 3]'

class Person:
    def __str__(self):
        return "Person对象"

print(str(Person()))   # 'Person对象'
```

注意事项：

- 自定义类可以实现`__str__`方法定义转换行为
- 与`repr()`的区别：`str()`面向用户，`repr()`面向开发者
- 不会改变原对象，总是返回新字符串

list

功能：将可迭代对象转换为列表

示例：

```
print(list('abc'))         # ['a', 'b', 'c']
print(list((1,2,3)))       # [1, 2, 3]
print(list({1:'a',2:'b'})) # [1, 2]

# 字典处理
print(list({'a':1,'b':2}.items()))  # [('a', 1), ('b', 2)]
```

注意事项：

- 字典转换时默认只获取键
- 字符串会被拆分为字符列表
- 生成器会被消耗（只能迭代一次）
- 集合转换时不会保证有序

set

功能：将可迭代对象转换为字典

示例略，与`list`类似，但是会自动去重。

§1.3.1 数学计算函数

abs

功能：返回数字的绝对值

示例：

```
print(abs(-10))      # 10
print(abs(3.14))     # 3.14
print(abs(3+4j))     # 5.0 (复数的模)

# 自定义对象
class Distance:
    def __abs__(self):
        return 100

print(abs(Distance()))  # 100
```

注意事项：

- 复数返回的是模长
- 自定义类可实现`__abs__`方法支持abs操作
- 不会改变原值，总是返回新值

divmod

功能：返回商和余数的元组

示例：

```
print(divmod(10, 3))    # (3, 1)
print(divmod(10.5, 3))  # (3.0, 1.5)
print(divmod(-10, 3))   # (-4, 2)

# 实际应用：分页计算
total_items = 101
items_per_page = 10
pages, remainder = divmod(total_items, items_per_page)
if remainder:
    pages += 1
print(f"需要{pages}页")  # 11页
```

注意事项：

- 结果符号与除数一致
- 浮点数计算可能有精度问题
- 除数不能为0（`ZeroDivisionError`）

pow

功能：幂运算

示例：

```
print(pow(2, 3))        # 8
print(pow(2, 3, 5))     # 3 (2^3 % 5)
print(pow(2, -1))       # 0.5
print(pow(2, 0.5))      # 1.414... (平方根)

# 与**运算符比较
print(2 ** 3 ** 2)      # 512 (右结合)
print(pow(2, pow(3, 2))) # 512
```

注意事项：

- 三参数形式比`x ** y % z`更高效
- 负指数返回浮点数结果
- 大数计算时注意性能

§1.3.2 函数式编程工具

map

功能：对可迭代对象应用函数

示例：

```
# 基本使用
numbers = [1, 2, 3]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # [1, 4, 9]

# 多参数
a = [1,2,3]
b = [4,5,6]
result = map(lambda x,y: x+y, a, b)
print(list(result))   # [5, 7, 9]

# 批量赋值
a, b, c = map(int, input().split())

# 实际应用：清洗数据
prices = ["$10.5", "$20", "$15.99"]
clean_prices = map(lambda p: float(p[1:]), prices)
print(list(clean_prices))  # [10.5, 20.0, 15.99]
```

注意事项：

- 惰性求值，节省内存
- 函数参数数量必须与输入序列匹配

- 比列表推导式更适用于复杂函数
- 返回的结果为`map`类型（迭代器）而非`list`或`tuple`，没有`__str__`函数（即无法直接利用`print`输出其中内容），也无法使用`[]`进行索引，也导致了其只能使用一次。

filter

功能：过滤序列中的元素

示例：

```
# 过滤偶数
numbers = [1, 2, 3, 4, 5]
even = filter(lambda x: x%2 == 0, numbers)
print(list(even))  # [2, 4]

# 过滤None值
data = [1, None, 3, None, 5]
clean = filter(None, data)  # None函数过滤假值
print(list(clean))  # [1, 3, 5]

# 实际应用：数据清洗
ages = [23, -5, 10, 150, 0]
valid_ages = filter(lambda a: 0 < a < 120, ages)
print(list(valid_ages))  # [23, 10]
```

注意事项：

- 函数返回`True`保留元素，`False`过滤元素
- 比列表推导式`[x for x in iter if condition]`更适用于复杂条件
- 同样返回迭代器，只能消费一次

reduce

功能：对序列进行累积计算

示例：

```
from functools import reduce

# 计算乘积
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 24

# 拼接字符串
words = ['Python', 'is', 'awesome']
sentence = reduce(lambda x,y: f"{x} {y}", words)
print(sentence)  # "Python is awesome"

# 初始值
sum_with_init = reduce(lambda x,y: x+y, [1,2,3], 10)
print(sum_with_init)  # 16 (10+1+2+3)
```

常见问题：

- 需要从`functools`导入
- 空序列无初始值会报`TypeError`
- 复杂操作可能降低可读性

§1.3.3 系统交互函数

open

功能：打开文件并返回文件对象

示例：

```
# 基本读写
with open('test.txt', 'w', encoding='utf-8') as f:
    f.write('Hello\nWorld')

with open('test.txt', 'r', encoding='utf-8') as f:
    print(f.read())  # Hello\nWorld

# 二进制模式
with open('image.jpg', 'rb') as f:
    data = f.read()

# 追加模式
with open('log.txt', 'a') as f:
    f.write('New log entry\n')
```

常见问题：

1. 文件路径问题（相对/绝对路径）
2. 编码问题（特别是Windows平台）
3. 忘记关闭文件导致资源泄漏

注意事项：

- 始终使用`with`语句确保文件关闭
- 明确指定编码（推荐`utf-8`，但是有的时候也需使用`gbk`或`ansi`）
- 处理大文件时使用逐行读取

print

功能：打印输出

示例：

```python
# 基本输出
print("Hello", "World")  # Hello World

# 输出到文件
with open('output.txt', 'w') as f:
    print("Save to file", file=f)

# 控制分隔符和结束符
print("a", "b", "c", sep=", ", end="!\n")  # a, b, c!
```

参数

- `sep`：用于控制多个值之间的分隔符，默认为空格。例如：

```python
print("苹果", "香蕉", "橙子")              # 输出：苹果 香蕉 橙子（默认用空格分隔）
print("苹果", "香蕉", "橙子", sep=", ")    # 输出：苹果, 香蕉, 橙子（用逗号+空格分隔）
print("2023", "12", "31", sep="-")       # 输出：2023-12-31（用减号分隔，适合日期）
```

- `end`：制打印结束时的符号，默认为换行符`\n`。例如：

```python
print("你好", end="")          # 输出：你好（不换行，下次打印会接在后面）
print("世界")                  # 输出：你好世界（两行合并了）

print("加载中", end="...")      # 输出：加载中...（末尾用...代替换行）
print("完成")                  # 输出：加载中...完成（接在同一行）
```

注意事项

- 大量打印可能影响性能
- 调试时可以使用`pprint`模块美化输出

input

功能：获取用户输入

示例：

```python
# 基本输入
name = input("请输入你的名字: ")
print(f"你好, {name}!")

# 类型转换
age = int(input("请输入你的年龄: "))
print(f"你明年将{age+1}岁")

# 安全输入
while True:
    try:
        num = float(input("请输入数字: "))
        break
    except ValueError:
```

```
        print("输入无效，请重试")
```

注意事项：

- `input`函数的返回值是字符串，而不会自动识别为int、float等。
- `input`函数会且仅通过换行符截断。若需对于空格、逗号等另行截断，则可使用`split`函数。

§1.3.4 其他函数

len

功能：获取对象容器的元素数目

示例与注意事项：详见§1.4.2

type

功能：获取对象类型或动态创建类

示例：

```
# 查看类型
print(type(10))         # <class 'int'>
print(type([]))         # <class 'list'>

# 动态创建类
MyClass = type('MyClass', (), {'x': 10})
obj = MyClass()
print(obj.x)            # 10

# 与isinstance()区别
class Parent: pass
class Child(Parent): pass
print(type(Child()) == Parent)    # False
print(isinstance(Child(), Parent)) # True
```

注意事项：

- 通常使用isinstance()进行类型检查更灵活
- 元类编程的高级特性
- 动态创建类可能影响代码可读性

isinstance

功能：检查对象是否是类或其子类的实例

示例：

```
# 基本使用
print(isinstance(10, int))      # True
print(isinstance('a', str))     # True

# 多类型检查
print(isinstance([], (list, tuple)))  # True

# 继承检查
class A: pass
class B(A): pass
b = B()
print(isinstance(b, A))         # True
```

注意事项：

- 比type()更推荐用于类型检查
- 可以检查多个类型
- 支持抽象基类检查

range

功能：用于生成不可变数字序列的内置函数，广泛用于循环控制和序列生成。

示例：

```
# range(stop): 生成从 0 到 stop-1 的整数序列
# 生成 0-4 的序列
for i in range(5):
    print(i, end=' ')  # 输出: 0 1 2 3 4

# 等价于
numbers = list(range(5))
print(numbers)  # [0, 1, 2, 3, 4]

# range(start, stop): 生成从 start 到 stop-1 的整数序列
# 生成 5-9 的序列
for i in range(5, 10):
    print(i, end=' ')  # 输出: 5 6 7 8 9

# 负值范围
for i in range(-5, 0):
    print(i, end=' ')  # 输出: -5 -4 -3 -2 -1

# range(start, stop, step): 生成从 start 到 stop-1 (正步长)或 stop 到 start+1 (负步长), 步长为
step 的序列
# 步长为2
for i in range(0, 10, 2):
    print(i, end=' ')  # 输出: 0 2 4 6 8

# 负步长
for i in range(10, 0, -1):
    print(i, end=' ')  # 输出: 10 9 8 7 6 5 4 3 2 1
```

注意事项：

- step 不能为 0（ValueError）
- 负步长时 start 应大于 stop
- 步长会影响序列是否包含 stop 值

§1.4 内置数据类型

Python内置了丰富的数据类型，这些类型构成了Python编程的基础。本章将详细介绍这些类型及其常用操作。

§1.4.0 数值类型(int，float，complex)

Python支持三种基本数值类型：

1. 整型(int)：任意大小的整数，如`42`、`-17`
2. 浮点型(float)：双精度浮点数，如`3.14`、`-0.001`
3. 复数(complex)：由实部和虚部组成，如`1+2j`

int

`int`代表整数。在Python中，`int`类型与C++、Java等编程语言不同，并没有位数限制。换言之，并不像C++和Java一样存在最大值`2147483647`，也就没有对应存在的`long long`类型。

float

对于`float`类型即浮点数，常在各种编程语言中用于表示小数。关于浮点数的具体二进制表示方法，此处不做赘述。如想了解，可以参考：IEEE 754 - 2019 浮点算数标准 | 知乎：https://zhuanlan.zhihu.com/p/480834719

我们只需注意：`float`所表示的数值可能存在误差。其核心原因在于很多十进制小数（如：`0.1`）在二进制下是无限循环小数，而计算机存储浮点数时只能存储有限小数，所以就会导致无限循环小数被截断，导致数值误差。这也就导致了经典的`0.1+0.2==0.30000000000000004`的结果。

> **Tip**
>
> 在日常开发与应用中，对于确定小数位数（如价格等）且数据以加减运算为主的时候，推荐直接以`int`存储数值，再在人机交互时加上小数点，可以有效避免上述误差。

complex

对于`complex`类型，基本字面量形式为`a+bj`，其中`a`、`b`均为`int`或`float`，分别表示此复数的实部和虚部。换言之，将此表示中的`j`换为`i`，则其就与数学中的复数表示基本一致。

> **Important**
>
> 在数学中，可直接使用i表示虚数单位，但是在Python中，必须使用`1j`（而`j`会被认作对象名称）。
>
> 虽然字面量形式中`a`与`b`都是`int`或`float`，但你也可以使用`complex`类型。使用`complex`类型时，其结果与数学下将之化简后相同。

§1.4.1 布尔类型(bool)

布尔类型只有两个值：

- `True`（真）
- `False`（假）

布尔运算遵循以下规则（具体见§1.1.2中的逻辑运算符一节）：

```
True and False  # False
True or False   # True
not True        # False
```

通常来说，在加减乘除等运算中，`True`等价于`1`，`False`等价于`0`。

我们需知道：比较运算符的返回值是`bool`类型。

§1.4.2 容器(list，tuple，set，dict，str)

在Python中，常用的容器有`list`，`tuple`，`set`，`dict`，`str`。他们具有很多的共性，但是也有一些区别。

这些容器都有一个特性，便是其大小，即其中元素数目。可以通过`len`函数获取元素数目，如：

```
l1 = [1, 2, 3]
l2 = ['abc', (1, 3), 2, 4]
t1 = (1, 3, 4)
t2 = ([1, 1, 'abc'], 'h', 'e', 996)
d1 = {'hello': 'world'}
d2 = {'music games': ['Phigros', 'Chunithm', 'Arcaea'], 'other games': ['Minecraft', 'Poly
Bridge']}
s1 = {1, 1, 4, 5, 1, 4} # 等价于{1, 4, 5}
s2 = {'hallo', 'ciallo~'}
st = 'hello world'

print(len(l1), len(l2), len(t1), len(t2), len(d1), len(d2), len(s1), len(s2), len(st))
# 输出: 3 4 3 4 1 2 3 2 11
```

在具体介绍之前，我们首先要明确：在Python及大部分编程语言中，对于存在下标的容器，其下标是从0开始标的。在Python中，容器`list`、`tuple`、`str`均支持利用下标进行索引。索引方式如下指出，不再在后文分别介绍。

```
l = [1, 2, 3]
t = (1, 2, 3)
s = 'abc'

# 利用索引读取元素
print(l[0])     # 1, 即输出l中的第一个元素
print(t[0])     # 1, 即输出l中的第一个元素
print(s[0])     # a, 即输出s中的第一个字符
print(l[3])     # IndexError: list index out of range

# Python特色: 负数索引
# 在Python中, 负数索引X[-a]等价于X[len(X)-a]
```

```
print(s[-1])    # c, 即输出s中的最后一个字符
print(s[-2])    # b, 即输出s中的倒数第二个字符
print(s[-3])    # a, 即输出s中的倒数第三个字符
print(s[-4])    # IndexError: string index out of range
# 由上面可以知道, 容器X所能正常使用的索引应当为整数且在范围[-len(X), len(X))

# Python特色: 切片
# 在Python中, 可以使用形如X[a:b]或X[a:b:c]等被称为“切片”的特殊索引获取容器的子集
# 一般而言, X[a:b]等价于[X[i] for i in range(a, b)], X[a:b:c]等价于[X[i] for i in range(a, b, c)], 具体例如下
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(nums[1:4])    # 输出: [1, 2, 3]
print(nums[4:1:-1]) # 输出: [4, 3, 2]
print(nums[1:4:2])  # 输出: [1, 3]
print(nums[1:5:2])  # 输出: [1, 3]
print(nums[2:2])    # 输出: []
# 除此之外, a, b, c也可以分别省略. a, b, c分别默认值为0, len(X), 1. 具体例如下
print(nums[:4])     # 输出: [0, 1, 2, 3]
print(nums[4:])     # 输出: [4, 5, 6, 7, 8, 9]
print(nums[:])      # 输出: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(nums[::2])    # 输出: [0, 2, 4, 6, 8]
print(nums[1::2])   # 输出: [1, 3, 5, 7, 9]
print(nums[::-1])   # 输出: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# tuple与str均为不可变序列; list为可变序列
# 可以对list通过如下方法进行元素修改
l[0] = 3        # 得到[3, 2, 3]
t[0] = 3        # 报错: TypeError: 'tuple' object does not support item assignment
s[0] = 'a'      # 报错: TypeError: 'str' object does not support item assignment
# 同时, list也可以通过切片进行修改序列, 如下:
# 替换切片部分
nums[2:5] = [20, 30, 40]
print(nums)  # 输出: [0, 1, 20, 30, 40, 5, 6, 7, 8, 9]
# 删除切片部分
nums[2:5] = []
print(nums)  # 输出: [0, 1, 5, 6, 7, 8, 9]
# 插入元素
nums[2:2] = [2, 3, 4]
print(nums)  # 输出: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

以下对这些容器分别进行介绍。

列表(list)

列表是可变的序列类型，其中的元素类型并不用保持一致。例：

```
# 基本操作
lst = [1, 2, 3]
lst.append(5)    # 在列表最后一项添加元素5，列表变为[1, 4, 3, 5]
lst.pop()        # 返回最后一项5并将其删除，列表变为[1, 4, 3]

# 列表推导式，详见§1.7.6
[x*2 for x in range(5)]   # [0, 2, 4, 6, 8]
```

元组(tuple)

元组是不可变的序列：

```
t = (1, 2, 3)
print(t[0])      # 1
# t[0] = 4       # 报错，元组不可变
```

字典(dict)

字典是键值对集合：

```
d = {'name': 'Alice', 'age': 20}
d['age'] = 21      # 更新值
d['city'] = 'NY'   # 添加新键与值
del d['name']      # 删除键与值

d = {1: 2, 3: 2}
d.keys()           # 得到d的所有键的容器，类型为dict_keys，类似于list但不能使用下标索引
print(d.keys())    # dict_values([2, 2])
d.values()         # 得到d的所有值的容器，类型为dict_values，类似于list但不能使用下标索引
print(d.values())  # dict_values([2, 2])
d.items()          # 得到d的所有键值对的容器，类型为dict_items，类似于每个元素都是二元元组的list，但不能使用下标索引
print(d.items())   # dict_items([(1, 2), (3, 2)])
# 以上几个函数均常用于for循环中，如：
for key in d.keys(): ...
for value in d.values(): ...
for key, value in d.items(): ...
```

集合(set)

集合是无序不重复元素集：

```
s = {1, 2, 3}
s.add(4)       # 向集合中添加元素4，得到{1, 2, 3, 4}
s.add(4)       # 向集合中添加重复元素，不会对其造成改变. 故得到{1, 2, 3, 4}
s.remove(2)    # 从集合中移除元素2，得到{1, 3, 4}
s.remove(2)    # 从集合中移除不存在的元素，报错：KeyError: 2
```

> **Tip**

> 在C++的STL中的集合set不同，Python中的set并不会自动排序，而C++中的set为有序（插入/删除元素后自动排序）的。

字符串(str)

字符串是不可变的字符序列，支持以下的操作：

```
# 基本操作
s = "Hello"
print(s[1])       # 'e'（索引）
print(s[1:4])     # 'ell'（切片）
print(len(s))     # 5（长度）

# 常用方法
"hello".upper()                 # 将所有英文字母大写，得到'HELLO'
"HELLO".lower()                 # 将所有英文字母小写，得到'hello'
" hello ".strip()               # 删除字符串两端的空白字符，包括空格(' ')、制表符('\t')和换行符
('\n')。得到'hello'

# split(sep=None, maxsplit=-1)函数：可以根据参数sep(str类型)将字符串分割，得到的每个子串按照顺序存储在
返回的list中。
# 分割用字符串不限长度，在分割后分割用字符串不会存在于返回值中
# sep为默认值(None)时，将会依空格(' ')、制表符('\t')和换行符('\n')进行分割
# maxsplit指分割的最多次数。换言之，最多分割出来maxsplit+1个子串
"a,b,c".split(',')              # ['a', 'b', 'c']
'1\t2\n3 4'.split()             # ['1', '2', '3', '4']
'1\t2\n3 4'.split(maxsplit=1)   # ['1', '2\n3 4']

# join(iterable)函数：将元素均为str的容器保持顺序地通过此字符串进行连接，例如：
'-'.join(['a','b','c'])         # 'a-b-c'
'-d-'.join(['a','b','c'])       # 'a-d-b-d-c'
'-'.join('hello')               # 'h-e-l-l-o'
'-'.join({'1','2','a','bbb'})   # set类型中元素顺序与对set类型使用str函数得到的结果顺序相同。得到'a-
1-2-bbb'
'-'.join({'1':'2', '3':'4'})    # dict类型直接传入时，将视作传入{ ... }.keys()。得到'1-3'
```

§1.5 常用内置库与其中函数

Python标准库提供了丰富的模块，本节介绍最常用的几个。

§1.5.0 math模块

数学运算函数：

```
import math

math.pi             # 3.141592653589793
math.e              # 2.718281828459045

math.sqrt(x)        # 平方根(返回float)
math.cbrt(x)        # 立方根(返回float)
math.pow(x, y)      # 乘方，换言之x**y
```

```
math.prod(iterable, start=1)    # 对iterable迭代，以start为初始值进行累乘，类似于内置函数sum
math.sumprod(p, q)  # 对于等长的迭代器p与q，返回其中对应项之积的和，即sum([i*j for i, j in zip(p, q, strict=True)])

math.factorial(x)   # 阶乘(x为自然数)
math.gamma(x)       # Gamma Γ函数(阶乘推广)
math.comb(n, k)     # 组合数，等于nCk(Casio Classwiz表达形式)，换言之n!/(k!*(n-k)!) if k <= n else 0
math.perm(n, k=None)# 排列数，等于nPk(Casio Classwiz表达形式)，换言之n!/(n-k)! if k <= n else 0. 特别的，k=None时等于n!
                    # comb与perm中的n与k都应为自然数

math.cos(x) / sin(x) / tan(x) / cosh(x) / sinh(x) / tanh(x)     # 三角/双曲函数（弧度制）
math.acos(x)/ asin(x)/ atan(x)/ acosh(x)/ asinh(x)/ atanh(x)    # 反三角/反双曲函数（弧度制）
math.atan2(y, x)    # 同时考虑y与x正负的反正弦函数. 换言之: 返回复数x+yi(Python中表达为x+yj)的辐角主值
math.degrees(x)     # 将弧度制角x转换为角度值
math.radians(x)     # 将角度值角x转换为弧度制

math.ceil(x)        # 上取整，即大于等于x的最小整数
math.floor(x)       # 下取整，即小于等于x的最大整数

math.gcd(*integers) # 多个整数的最大公约数(正)
math.lcm(*integers) # 多个整数的最小公倍数(正)

math.log(x, base=e) # 取对数，默认底数为e
math.log10(x)       # 取常用对数，即math.log(x, 10)
math.log2(x)        # 取二为底的对数，即math.log(x, 2)

math.dist(p, q)     # 求点p与q之间的欧式距离，即sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q))).
                    # 其中p与q为等长的序列/迭代器，为此点的坐标coordinates构成的序列
math.hypot(*coordinates)    # 求点到原点的距离，即math.dist(coordinates, [0]*len(coordinates))
```

§1.5.1 random模块

随机数生成：

```
import random

random.random()         # [0,1)随机浮点数
random.uniform(a,b)     # [a,b]随机浮点数
random.randint(a,b)     # [a,b]随机整数
random.getrandbits(k)   # 返回具有k个随机位的Python整数
random.randbytes(n)     # 生成n个随机字节

random.randrange(stop)  # 从range(stop)中随机选择一个元素
random.randrange(start, stop[, step])   # 从range(start, stop[, step])

random.choice(seq)      # 从非空序列seq返回一个随机元素
```

```
random.choices(population, weights=None, k=1)   # 从序列population中返回大小为k的元素列表(可重复选择, weights为相对权重列表)
random.sample(population, k)                    # 从序列population中选择k个唯一元素(无重复)
random.shuffle(x[, random])                     # 将可变序列x打乱. random为返回(0.0, 1.0)数的函数

# 以下为生成符合特定分布随机数的函数, 具体分布函数表达式请自行查阅数学书
random.triangular(low, high, mode)      # 三角形分布, low为下限(默认0), high为上限(默认1), mode为众数(默认中点)
random.betavariate(alpha, beta)         # Beta分布
random.expovariate(lambd)               # 指数分布, lambd为1.0除以此分布的期望(必须非零)
random.gammavariate(alpha, beta)        # Gamma分布
random.gauss(mu, sigma)                 # 高斯分布, mu为期望, sigma为标准差
random.normalvariate(mu, sigma)         # 正态分布, mu为期望, sigma为标准差
random.lognormvariate(mu, sigma)        # 对数正态分布, mu为期望, sigma为标准差
random.vonmisesvariate(mu, kappa)       # 冯·米塞斯分布(圆形正态分布), mu为平均角度(弧度), kappa为集中度参数
random.paretovariate(alpha)             # 帕累托分布
random.weibullvariate(alpha, beta)      # 威布尔分布
```

§1.5.2 datetime模块

日期时间获取与处理

```
from datetime import datetime as dt
import datetime.UTC
# 以下函数返回值均为datetime模块中的特定类型对象. 如需输出转换为str, 则需使用strftime函数以格式化代码, 具体参见后文表格
dt.today()                          # 返回本地datetime(无时区信息)
dt.now(tz=None)                     # 返回本地日期时间, 其中tz参数跟的是UTC时区, 可以填入datetime.UTC以设定为当前系统UTC
dt.fromtimestamp(timestamp)         # 从时间戳返回datetime
dt.utcfromtimestamp(timestamp)      # 从时间戳返回UTC datetime
dt.romisoformat(date_string)        # 从ISO格式字符串返回datetime
dt.combine(date, time)              # 组合date和time为datetime

# 以下函数均为datetime类型对象的实例函数
date()                              # 返回date部分
time()                              # 返回time部分
replace(year, month, day, hour, minute, second, microsecond, tzinfo)
                                    # 返回替换后的新datetime对象而不改变自身值. 参数默认值均为原有值, 略去
timetuple()                         # 返回time.struct_time
timestamp()                         # 返回POSIX时间戳
weekday()                           # 返回星期几（0-6, 0是周一）
isoweekday()                        # 返回星期几（1-7, 1是周一）
isocalendar()                       # 返回(year, week number, weekday)元组
isoformat(sep='T')                  # 返回ISO格式的字符串
strftime(format)                    # 返回自定义格式的字符串

# 一实例如下:
now = datetime.now()
```

```
print(now.strftime("%Y-%m-%d"))  # 格式化输出
```

特别的，上文中ISO格式指ISO 8601格式，具体格式文档参见ISO 8601 | 百度百科：https://baike.baidu.com/item/ISO%208601/。其中`isoformat`以默认值输出的结果形如：`2025-06-18T22:00:30.702635`（这是编写此节的时间）。具体相关内容此处不作过多叙述，可自行查阅相关官方文档。

日期时间的格式化IO

```
from datetime import datetime

# strftime为datetime类型对象的实例函数，而strptime则是datetime类型的类型函数/静态函数(此处并不确定)
# strftime可按照固定格式将datetime类型转换为字符串，strptime可按照固定格式将字符串转换为datetime
# 具体应用实例如下
now = datetime.now()
print(now.strftime("%Y-%m-%d"))  # 格式化输出
parsed = datetime.strptime('2023年05月15日', '%Y年%m月%d日') # 格式化读取
print(parsed)  # 2023-05-15 00:00:00
```

而这些格式化参数占位符具体含义如下：

代码	含义	示例
%Y	4位数年份	2085
%y	2位数年份85	85
%m	月份（01-12）	11
%d	日（01-31）	11
%H	小时（00-23）	14
%I（大写i）	小时（01-12）	02
%M	分钟（00-59）	14
%S	秒（00-59）	19
%f	微秒（000000-999999）	19810
%A	完整星期名称	Sunday
%a	简写星期名称	Sun
%B	完整月份名称	November
%b	简写月份名称	Nov
%j	一年中的第几天（001-366）	315
%U	一年中的第几周（00-53，周日为一周开始）	45
%W	一年中的第几周（00-53，周一为一周开始）	45
%w	星期几（0-6，0是周日）	0

代码	含义	示例
%Z	时区名称	UTC
%z	UTC偏移量	+0800

其中星期名称对应表如下：

完整名称 (Full Name)	简写名称 (Abbreviation)
Monday	Mon
Tuesday	Tue
Wednesday	Wed
Thursday	Thu
Friday	Fri
Saturday	Sat
Sunday	Sun

月份名称对应表如下：

完整名称 (Full Name)	简写名称 (Abbreviation)	备注
January	Jan	
February	Feb	
March	Mar	
April	Apr	
May	May	无简写，与完整名称相同
June	Jun	
July	Jul	
August	Aug	
September	Sep	或 `Sept`（非Python标准）
October	Oct	
November	Nov	
December	Dec	

日期时间差值计算

可以利用`datetime`中的`timedelta`对象进行日期差值计算。直接对`datetime.datetime`对象进行详见，即可得到`datetime.timedelta`对象，从而得到结果，例如：

```
from datetime import datetime
# 计算两个日期时间的差值
dt1 = datetime(2023, 5, 15, 10, 30)
dt2 = datetime(2023, 5, 16, 12, 45)
delta = dt2 - dt1  # 类型: datetime.timedelta
print(delta)  # 输出: 1 day, 2:15:00 (相差1天2小时15分钟)
```

`timedelta`对象有以下常用属性：

- `.days` → 相差的天数（整数）
- `.seconds` → 相差的秒数（不包括天数，0-86399）
- `.microseconds` → 相差的微秒数
- `.total_seconds()` → 总相差秒数（包括天数）

接上例子，可有下例：

```
print(delta.days)  # 1
print(delta.seconds)  # 8100 (2小时15分钟 = 2*3600 + 15*60)
print(delta.total_seconds())  # 98100.0 (1天=86400秒 + 8100秒)
```

`datetime.timedelta`可以与`datetime.timedelta`进行加减运算。也可以与`int`、`float`类型进行乘法、除法运算，即进行与你所想一样的时间倍增或倍减。

而`datetime.timedelta`对象也可以与`datetime.datetime`进行加减运算，分别代表向后或向前推进时间。接上例子，可有下例：

```
print(delta * 2)  # 2 days, 4:30:00
print(delta / 2)  # 13:07:30
print(delta + delta)  # 2 days, 4:30:00
print(dt1 + delta)  # 2023-05-16 12:45:00
print(dt1 + delta * 2)  # 2023-05-17 15:00:00
```

§1.5.3 os模块

os模块常用于Python与操作系统的交互。

> ⚠ **Warning**
>
> 由于os库主要与操作系统进行交互，所以在不同平台之间存在一定的差异，使代码可迁移性降低。所以如非必要，尽量避免使用os库。如必须使用，请至少对Unix-like和Windows两个平台进行分别处理（除非你能保证此代码的运行环境），即使用以下的分类讨论代码：

```
if os.name == 'nt':
# Windows特定代码
else:
# Unix特定代码
```

实际上，`os.name`的返回值与系统版本对应如下：

- `'posix'`：Unix-like系统
- `'nt'`：Windows
- `'java'`：Jython（JVM虚拟机）

以下介绍在Windows平台下，os库常用的函数。具体实例此处不作列出，如需了解请自行尝试。

文件和目录操作

```
# 文件操作
os.rename(src, dst)                 # 重命名文件或目录
os.remove(path)                     # 删除文件
os.unlink(path)                     # 删除文件(与remove相同)
os.startfile(path[, operation])     # 使用关联程序打开文件
os.path.getsize(path)               # 获取文件大小(字节)
os.path.split(path)                 # 分割路径为目录和文件名
os.path.splitext(path)              # 分割路径为文件名和扩展名

# 目录操作
os.mkdir(path)                      # 创建单个目录
os.makedirs(path)                   # 递归创建目录
os.rmdir(path)                      # 删除空目录
os.removedirs(path)                 # 递归删除空目录
os.listdir(path)                    # 列出目录内容
os.getcwd()                         # 获取当前工作目录
os.chdir(path)                      # 改变当前工作目录

# 路径操作
os.path.join(path1[, path2[, ...]]) # 连接路径组件
os.path.abspath(path)               # 返回绝对路径
os.path.basename(path)              # 返回路径的基本名称
os.path.dirname(path)               # 返回路径的目录名
os.path.exists(path)                # 检查路径是否存在
os.path.isfile(path)                # 检查是否为文件
os.path.isdir(path)                 # 检查是否为目录
```

系统信息和环境

```
os.name                             # 操作系统名称(Windows下为'nt')
os.environ                          # 系统环境变量字典
os.getenv(key[, default])           # 获取环境变量值
os.putenv(key, value)               # 设置环境变量值
os.system(command)                  # 执行系统命令
os.popen(command[, mode[, bufsize]])# 执行命令并返回文件对象
```

进程管理

```
os._exit(n)                          # 直接退出进程(不清理)
os.getpid()                          # 获取当前进程ID
os.getppid()                         # 获取父进程ID
os.kill(pid, sig)                    # 向进程发送信号(Windows上功能有限)
os.spawn...(...)                     # spawn函数家族，不常用，故不分别列出，详见Python官方文档
```

文件描述符操作

```
os.open(path, flags[, mode])         # 低级文件打开
os.close(fd)                         # 关闭文件描述符
os.read(fd, n)                       # 从文件描述符读取
os.write(fd, str)                    # 向文件描述符写入
os.dup(fd)                           # 复制文件描述符
os.dup2(fd, fd2)                     # 复制文件描述符到指定编号
```

其他实用功能

```
os.urandom(n)                        # 生成加密安全的随机字节
os.times()                           # 返回进程时间信息
os.umask(mask)                       # 设置默认权限掩码
os.fsync(fd)                         # 强制将文件写入磁盘
```

Windows特有功能

```
os.path.expandvars(path)             # 展开路径中的环境变量
os.path.normpath(path)               # 规范化Windows路径(处理反斜杠)
os.path.splitdrive(path)             # 分割驱动器和路径
```

> **Important**
> 在Windows上使用路径时，建议使用原始字符串`r'C:\path'`或双反斜杠`'C:\\path'`以避免转义问题。

§1.5.4 sys模块

sys模块用于Python标准库中与Python解释器交互的重要模块。一些常用函数如下。

命令行参数

`sys.argv`：包含命令行参数的列表，其中每一元素内容：

- `sys.argv[0]`：脚本名称
- `sys.argv[1]`：第一个参数，以此类推

例：

```
# 文件名: test.py
import sys
print("脚本名:", sys.argv[0])
print("参数:", sys.argv[1:])
```

```
>>> python test.py
脚本名: test.py
参数: []
>>> python test.py 1
脚本名: test.py
参数: ['1']
>>> python test.py 1 2 3
脚本名: test.py
参数: ['1', '2', '3']
>>> python test.py 1 2 3 -s 2
脚本名: test.py
参数: ['1', '2', '3', '-s', '2']
```

标准输入/输出流

- `sys.stdin`：标准输入流
- `sys.stdout`：标准输出流
- `sys.stderr`：标准错误流

例：

```
import sys
sys.stdout.write("直接写入标准输出\n")
error_msg = "错误信息"
sys.stderr.write(f"错误: {error_msg}\n")
```

系统信息

- `sys.platform`：操作系统平台标识符
- `sys.version`：Python版本信息
- `sys.maxsize`：最大整数值（Python整数大小，在Python3中不必考虑）

例：

```
import sys
print(f"运行平台: {sys.platform}")
print(f"Python版本: {sys.version}")
```

退出程序

`sys.exit([arg])`：退出Python程序
    0表示成功退出（默认值即0）
    非0值表示异常退出

甚至可以使用`sys.exit("sorry, goodbye!")`这类语句，即可在退出的同时输出其中字符串等（实际上只要此值不是整型，那么就会在退出程序前输出其中参数）。

例：

```
import sys
if error_condition:
    sys.exit(1)  # 异常退出
else:
    sys.exit('sorry, goodbye!')  # 输出sorry, goodbye!并退出
```

> **Important**
>
> 对比：`os._exit()`，`sys.exit()`，`exit()`，`quit()`：
>
> - `os._exit()`使程序直接退出，不抛出异常且不执行相关清理工作。常用在子进程的退出。
> - `sys.exit([arg])`退出程序可引发`SystemExit`异常，可以捕获异常执行些清理工作。一般主程序中使用此退出。
> - `exit()`/`quit()`抛出`SystemExit`异常。一般在交互式shell中退出时使用。

模块和导入系统

- `sys.modules`：已加载模块的字典
- `sys.builtin_module_names`：内置模块名称元组
- `sys.meta_path`：元路径查找器列表

例：

```
import sys
print("已加载模块:", list(sys.modules.keys()))
```

内存管理

- `sys.getsizeof(object)`：获取对象占用的内存大小(字节)
- `sys.getrefcount(object)`：获取对象的引用计数

例：

```
import sys
x = [1, 2, 3]
print(f"列表占用内存: {sys.getsizeof(x)} 字节")
```

异常处理

- `sys.exc_info()`：获取当前异常信息(类型,值,回溯)
- `sys.last_type`, `sys.last_value`, `sys.last_traceback`：最后未处理的异常

例：

```
import sys
try:
    1 / 0
except:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print(f"异常类型: {exc_type}")
```

线程和递归控制

- `sys.setrecursionlimit(limit)`：设置递归深度限制
- `sys.getrecursionlimit()`：获取当前递归深度限制
- `sys.settrace(tracefunc)`：设置全局跟踪函数
- `sys.setprofile(profilefunc)`：设置全局性能分析函数

其他实用功能

- `sys.implementation`：Python实现信息
- `sys.flags`：命令行标志信息
- `sys.thread_info`：线程实现信息
- `sys.hash_info`：哈希算法信息
- `sys.float_info`：浮点数实现信息
- `sys.int_info`：整数实现信息
- `sys.set_int_max_str_digits(n)`：设置`int`转`str`的位数上限

例：

```
import sys
print("浮点数信息:", sys.float_info)
print("哈希信息:", sys.hash_info)
```

> **Important**
>
> 注意，sys库的修改函数中，基本为直接修改Python解释器中的特定变量。所以在CPython中，大部分数值设定（如`sys.setrecursionlimit(limit)`、`sys.set_int_max_str_digits(n)`）函数中的数值存在C int的范围限制，即-2147483648至2147483647，或者说-2^{31}至$2^{31}-1$。事实上这些数值基本上取值应为正，故只需记住最大值：2147483647。

§1.6 面向对象入门

§1.6.0 什么是面向对象？

Python从设计之初就是一门面向对象的语言。面向对象编程（OOP）是一种编程思想，它把数据和操作数据的方法打包在一起，称为"对象"。本章将带你了解Python中面向对象的基本概念。

> **Tip**
>
> 初次学习时不必完全掌握所有细节，理解基本概念即可。如需深入学习，

类(Class)：用来描述具有相同属性和方法的对象的模板。比如“学生”可以是一个类。
对象：类的具体实例。比如“张三”是“学生”类的一个对象。
属性：对象具有的特征。比如学生的“姓名”、“学号”。
实例方法：对象可以执行的操作。比如学生的“选课”、“提交作业”。
类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
继承：一个派生类（derived class）继承基类（base class）的字段和方法。继承允许把派生类对象当作基类对象处理。例如：Dog类继承自Animal类，表示"Dog是一个Animal"的关系。
方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
实例化：创建一个类的实例，类的具体对象。

§1.6.1 类的基本结构

```python
class Student:
    '学生类的示例'
    student_count = 0  # 这是类变量，所有学生共享

    def __init__(self, name, id):
        self.name = name  # 实例变量
        self.id = id      # 实例变量
        Student.student_count += 1

    def show_info(self):
        print(f"学生姓名: {self.name}, 学号: {self.id}")

# 在使用实例方法时，应以形如 对象.函数(参数) 的形式进行运用
# 其中的参数，即约同于实例方法中的self删去后正常定义(指非实例方法/类方法)的函数
stu1=Student('李展鸣', 20240209)
stu1.show_info()  # 学生姓名: 李展鸣, 学号: 20240209
print(stu1.student_count, Student.student_count)  # 1 1

stu2=Student('王宇豪', 20242324)
stu2.show_info()  # 学生姓名: 王宇豪, 学号: 20242324
print(stu1.student_count, stu2.student_count, Student.student_count)  # 2 2 2
```

> **Tip**
> `学生类的示例` 是类的说明文档，可以用 `Student.__doc__` 查看。
> 建议放在类定义的第一行，使用单引号；或者使用双引号引导的多行字符串。

`__init__()`：这是类的构造函数，创建对象时会自动调用，即对于类型A，使用`A(...)`创建对象时，其中参数即相当于调用`__init__`的实例函数时的参数。

`self`：表示类的实例本身，相当于其他语言中的`this`。

> **Note**
> 虽然可以用其他名字代替`self`，但按照惯例请始终使用`self`，这样代码更易读。

§1.6.2 理解self参数

`self`代表类的实例，而不是类本身。看这个例子：

```
class Test:
    def show_self(self):
        print(self)
        print(self.__class__)

t = Test()
t.show_self()
```

输出结果：

```
<__main__.Test object at 0x...>
<class '__main__.Test'>
```

可以看到，`self`指向的是具体的对象实例。而若需获取类型，则可使用`__class__`属性。

§1.6.3 创建对象实例

```
# 创建两个学生对象，在上文中已有示例(接上代码):
stu1=Student('李展鸣', 20240209)
stu2=Student('王宇豪', 20242324)
```

在Python中，创建对象就像调用函数一样简单，不需要`new`关键字。

> **Tip**
>
> 你可以近似的认为，`李展鸣=Student('李展鸣', 20240209)`类似于执行了`Student.__init__(self, "李展鸣", 20240209)`并赋值给变量`李展鸣`。

§1.6.4 访问对象属性和方法

```
# 接上代码
stu1.show_info()  # 学生姓名: 李展鸣, 学号: 20240209
print(stu1.student_count, Student.student_count)  # 1 1

stu2.show_info()  # 学生姓名: 王宇豪, 学号: 20242324
print(stu1.student_count, stu2.student_count, Student.student_count)  # 2 2 2
```

除了访问，还可以可以动态修改对象属性：

```
stu1.name = '李昀翰'  # 修改姓名
stu1.grade = 'A'      # 添加新属性
```

常用属性操作函数（obj指代变量名，name指代属性名）如下：
`getattr(obj, name)`：获取属性
`hasattr(obj, name)`：检查属性是否存在
`setattr(obj, name, value)`：设置属性
`delattr(obj, name)`：删除属性

> **Tip**
> 如果不好理解，你可以认为Class是一个自定义函数和一个自定义变量的集合，并通过变量的形式调用。

§1.6.5 类的内置属性

每个类都有一些内置属性：

```
print(Student.__doc__)     # 类的文档字符串
print(Student.__name__)    # 类名
print(Student.__module__) # 类所在的模块
print(Student.__dict__)    # 类的属性字典
print(Student.__bases__)   # 类的所有父类构成元素(包含了一个由所有父类组成的元组)
```

例：

```
class Person:
    """这是Person类的文档字符串"""
    pass

class Student(Person):
    """这是Student类的文档字符串

    这个类代表一个学生，包含学生的基本信息"""

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def study(self, subject):
        return f"{self.name}正在学习{subject}"

# 使用内置属性
print(Student.__doc__)     # 输出: 这是Student类的文档字符串...
print(Student.__name__)    # 输出: Student
print(Student.__module__) # 输出: __main__ (如果在交互式环境或直接运行)
print(Student.__dict__)    # 输出: 类的属性字典
print(Student.__bases__)   # 输出: (<class '__main__.Person'>,)
```

§1.6.6 对象的生命周期

Python使用引用计数管理内存。当对象没有被引用时会被自动回收。可以定义`__del__`方法在对象销毁时执行清理：

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __del__(self):
        print(f"点({self.x}, {self.y})被销毁")

p = Point(1, 2)
del p  # 会运行__del__, 输出: 点(1, 2)被销毁
```

> **Important**
>
> 关于引用计数，此处不作介绍。如需了解其具体算法及弊端，可以参考：《【python】引用计数机制中循环引用可能带来的问题》_哔哩哔哩：https://www.bilibili.com/video/BV1vsNweAETc/

§1.6.7 类的继承

继承让我们可以创建新的类来扩展已有类的功能：

```python
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print(f"你好，我是{self.name}")

class Student(Person):
    def __init__(self, name, id):
        super().__init__(name)  # 调用父类的构造方法
        self.id = id

    def show_id(self):
        print(f"我的学号是{self.id}")

stu = Student("小明", "1001")
stu.say_hi()    # 继承自Person的方法
stu.show_id()   # Student自己的方法
```

> **Tip**
>
> `super()`用于调用父类的方法，从而达成续写父函数的目的。
>
> 也可以使用`父类名.方法名(self,...)`的方式（可用但并不推荐）。

§1.6.8 方法重写

子类可以重写父类的方法：

```python
class Animal:
    def speak(self):
        print("动物叫声")

class Dog(Animal):
    def speak(self):  # 重写父类方法
        print("汪汪汪")

dog = Dog()
dog.speak()  # 输出"汪汪汪"
```

§1.6.9 私有属性和方法

在属性/方法名前加单下划线`_`或两个下划线`__`表示私有：

```python
class Secret:
    __secret_code = "123"  # 私有属性

    def __get_code(self):  # 私有方法
        return self.__secret_code

    def show_code(self):
        print(f"密码是{self.__get_code()}")

s = Secret()
s.show_code()  # 正确
# s.__get_code()  # 错误，不能直接调用私有方法
```

> **Tip**
>
> 关于Python中属性/方法前的下划线含义如下：
>
> 单下划线`_name`表示受保护的属性/方法，仍可被外部直接访问，但在现代IDE（如VSCode，PyCharm）中，会对此种行为进行警告。
>
> 双下划线`__name`表示私有属性/方法，不可被外部直接访问。其会被Python解释器自动重命名。对于类型`X`的属性/方法`__a`，其会被重命名为`_X__a`。
>
> 双下划线开头和结尾`__name__`是特殊属性/方法，其中的特殊方法即魔法函数，参见§1.7.7。

§1.7 Python 部分高级语法、次常用函数与语法糖

这一章节中以Python中的一些高级语法及次常用函数、语法糖进行介绍。在这一段的教材中，将默认读者已经基本掌握Python的基本语法，并对一些编程的专有名词等有一定了解。

§1.7.0 二进制与位运算

在Python中的二进制与位运算与其他编程语言中类似。

二进制数的表达与转换

Python中可以利用`0b...`来用二进制定义int，其中`...`就表示所需要的二进制数，例子如下：

```
a = 0b1001 # a = 9
b = 0b0010010011100 # b = 1180
```

> ⚠ **Warning**
>
> 注意，以下的表达是不合法的：

```
>>> 0b114514
"""报错输出如下
  File "<stdin>", line 1
    0b114514
        ^
SyntaxError: invalid digit '4' in binary literal
"""

>>> 0b1.01
"""报错输出如下
  File "<stdin>", line 1
    0b1.01
       ^^^
SyntaxError: invalid syntax
"""
```

而Python中也可以用`bin`函数来将int转化为str类型的二进制文本，如：

```
>>> bin(1)
'0b1'
>>> bin(5)
'0b101'
```

> **Important**
>
> 请注意，转化后的`str`包括前缀`0b`，使用时需注意

> **Tip**
>
> 事实上，Python中除了二进制的直接定义与转换，还有八进制与十六进制的定义与转换，其表达与二进制类似：定义均为`0+字母+数`，而对应的转换函数进行转换后也会在字符串前面加上`0+字母`的前缀，具体如下：
>
> | 进制 | 二进制 | 八进制 | 十六进制 |
> | --- | --- | --- | --- |
> | 前缀 | 0b | 0o | 0x |
> | 转换函数 | bin | oct | hex |
>
> 具体例子略去，可自行在Python中尝试。

二进制运算

Python中的二进制运算与其他语言类似，具体含义如下（下表中的数均在二进制的表达下）：

运算名称	运算数学表达	Python表达	运算示例	等价表达
与（and）	两个位都为1时，结果才为1	a & b	0100 & 1101 = 0100	略
或（or）	两个位都为0时，结果才为0	a | b	0100 | 1101 = 1101	略
异或（xor）	两个位相同为0，相异为1	a ^ b	0100 ^ 1101 = 1001	略
取反（not）	按位0变1，1变0	~a	~0100 = 1011	略
左移	各二进位全部左移若干位，高位丢弃（Python中并不会丢弃），低位补0	a << b	1101 << 1 = 11010	a * (2 ** b)
右移	各二进位全部右移若干位，高位补0，低位丢弃	a >> b	1101 >> 1 = 110	a // (2 ** b)

与`+ - * / //`类似的，二进制运算（取反除外）也存在符合赋值运算符，即

```
a &= b 等价于 a = a & b
a |= b 等价于 a = a | b
a ^= b 等价于 a = a ^ b
a <<= b 等价于 a = a << b
a >>= b 等价于 a = a >> b
```

§1.7.1 Python中的复合语句

伪·三元运算符1

在C++中，很常见的一个语句就是三元判断运算符，如下：

```
a ? b : c
```

其含义就等价于

```
// C++
if (b) { return a; }
else { return c; }
```

而在Python中，有一个“三元运算符”，如下：

```
(c, b)[a]
```

其含义类似于

```
if a:
    return b
else:
    return c
```

> ⚠ **Warning**
>
> 注意，通常来说请不要使用这个三元运算符，因为其本质就是一个元组和一个索引，这也就导致了以下问题：
>
> 1. 如果b不是bool类型而是int类型、str类型等，则无法实现类似if判断的自动转化过程
> 2. 因为这是首先声明一个元组，再去进行索引，那么事实上其代码是这样的：
>
> ```
> ret_a = a()
> ret_c = c()
> if b:
>     return ret_a
> else:
>     return ret_c
> ```
>
> 这也就导致了无论b是`True`还是`False`，a和c如果是函数，那么都会运行，导致出期望之外的结果（翻译：Bug）。

总而言之，不推荐使用这种语法，但是在某些地方，在保证完全可行及必须的前提下，可以使用此语法进行代码压缩。

但是，这也给了我们另外一种思路：可以在key仅有int且数量较少的情况下，使用元组或列表来替换词典（但事实上这在大部分时候没有什么用，写出来也大概率是个防御性代码）。

伪·三元运算符2

除了上文运用元组和索引实现的伪·三元运算符，还有另一种利用逻辑运算符的运算顺序与返回特性“三元运算符”，如下：

```
a and b or c
```

其含义就等价于

```
if a:
    return b
else:
    return c
```

其相较于利用元组实现的三元运算符，因为`and`和`or`所具有的截断特性，所以并不会有类似上文错误运行的结果。

> ⚠ **Warning**
>
> 这种写法这也会导致一些问题。因为其对于`b`和`c`也会在一些时候进行布尔值判断，所以如果`and`后面跟的是假值，如`''`、`0`、`[]`等，那么所期望的`and`截断返回就不会进行，`or`会继续执行，导致可能的错误结果，例如：

```
age = 17
status = age >= 18 and "" or "未成年"  # 返回 "未成年"（正确）
age = 20
status = age >= 18 and "" or "成年"   # 返回 "成年"（错误！因为 "" 是假值）
```

所以说这种语法同样不推荐。

真·三元运算符与单行逻辑语句

正如上文所示，利用元组或逻辑运算符实现的三元运算符都会导致一些隐性的bug，所以我们下面介绍Python语法中的标准三元运算符。

相较于C++等其他语言的符号型三元运算符，Python中的三元运算符更为符合自然语言，具体形式如下：

```
a if c else b
```

其基本相当于

```
def f():
    if c:
        return a
    else:
        return b
f()
```

或者C++中的

```
c ? a : b
```

于是，我们也可以将此三元运算符称作“单行if-else语句”。

> **Tip**
>
> 对于上述`a if c else b`的语句中，b也可以再次使用“单行if-else语句”取代，且并不需要添加小括号来强调其整体性，如下：
>
> ```
> a if c else d if e else b
> ```
>
> 其也就等价于
>
> ```
> def f():
> if c:
>   return a
> else:
>   if e:
>       return d
>   else:
>       return b
> f()
> ```

> 或者说
>
> ```
> def f():
> if c:
>  return a
> elif e:
>  return d
> else:
>  return b
> f()
> ```
>
> 类似的，我们还可以继续进行类似的替换。但是，在“单行if-else语句”中，并没有直接支持`elif`语句。

除此之外，Python中还有另一个单行逻辑语句，即“单行for语句”或“生成器”，有例子如下：

```
[i for i in range(10)] # 等于[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   <list object>
(i for i in range(10)) # 得到<generator object>
{i for i in range(10)} # 等于{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}   <set object>
{i: (i, i+1) for i in range(3)} # 等于{0: (0, 1), 1: (1, 2), 2: (2, 3)}   <dict object>
```

其中圆括号得到的结果说明了单行for语句的本质是一个简洁的generator（生成器）声明，关于generator的相关内容见下文。

而上述第一、三、四行的语句，本质就是对于此generator进行了分别的类型转换。

> **Tip**
>
> 当你在使用“单行for语句”时，如果循环变量并不被使用，可以使用`_`来代替循环变量，且多个循环的循环变量可共用此变量，并不会冲突（看谁还敢说三段论的for更好）。

> **Warning**
>
> 请注意，如果需要在“单行if-else语句”中使用海象运算符，请尽量为海象运算符添加括号，以增加可读性（不然语句执行顺序会把人搞疯，同时也可能会导致一些奇怪的bug）。

并且，“单行for语句”与“单行if-else语句”也可以结合使用，且此时可以没有`else`语句，例如：

```
# 注意，在下面这些用法中，不能使用else语句
[i for i in range(10) if i % 2 == 0]    # 等于[0, 2, 4, 6, 8]
[i for i in range(10) if i % 2 == 0 for _ in range(3)]  # 等于[0, 0, 2, 2, 4, 4, 6, 6, 8, 8]
[i for i in range(10) for _ in range(2) if i % 2 == 0]  # 等于[0, 0, 2, 2, 4, 4, 6, 6, 8, 8]

# 而在以下用法中，则必须使用else语句
[i if i % 2 == 0 else 0 for i in range(10)] # 等于[0, 0, 2, 0, 4, 0, 6, 0, 8, 0]
```

Python中的分号（;）

虽然Python并不像其他语言一样，必须使用分号来分隔各个语句，但是你仍可以选择使用分号来在单行内运行多个语句，而且如果你想，你也可以在每一句话的结尾添加分号，然后就直接换行，这样的话就和没有分号是等价的。分号的常见示例如下：

```
x = 1; y = 2; print(x + y)  # 输出 3
if x > 0: print("正数"); x += 1
```

> **Tip**
>
> 请注意，对于`if`/`for`/`try`等冒号后接代码块的语句，如需使用单行复合结构，则只能有一个，否则会抛出`SyntaxError`。例如，以下语句就是错误的：
>
> ```
> if test1: if test2: print(x)
> ```
>
> 还要注意的是在这种情形下分号的绑定比冒号更紧密，因此在以下示例中，所有print要么都不执行，要么都执行:
>
> ```
> if x < y < z: print(x); print(y); print(z)
> ```

> **⚠ Warning**
>
> 根据PEP 8 | Style Guide for Python Code，分号的用法并不推荐使用。
>
> 通常来说，分号的使用场景通常为：
>
> 在Jupyter Notebook、命令行等交互式环境的使用中，快速执行多个操作
> 在极少数情况下，用分号以避免语法解析错误（如一行内混合语句和表达式）
> 临时测试
>
> 推荐使用多行语句或列表推导式、三元表达式等单行语法来代替分号。

§1.7.2 Python中的类型注释（Type Annotations）

在 Python 中，类型注释是一种用于指定变量、函数参数、返回值等类型的语法，目的是提高代码的可读性、可维护性，并借助静态类型检查工具（如 `mypy`）或IDE提前发现潜在的类型错误。Python 的类型注释是可选的，不会影响运行时行为（Python 仍然是动态类型语言），即即使在运行时变量/返回值出现了与标注类型不同的值，也不会抛出错误。

Python 的类型注释使用冒号（`:`）标注变量类型，箭头（`->`）标注函数返回值类型，格式如下：

```
变量名: 类型 = 值
def 函数名(参数: 类型) -> 返回值类型:
    ...
```

示例如下：

```python
# 变量类型注释
name: str = "Alice"
age: int = 25
is_student: bool = True
scores: list[float] = [90.5, 85.0, 77.5]
price: float = 9.99
data: bytes = b"binary"

# 函数参数和返回值注释
def greet(name: str) -> str:
    return f"Hello, {name}"
def add(a: int, b: int) -> int:
    return a + b
```

特别的，除基础类型int、float、str、bool、list、tuple、dict等的直接使用外，对于某些容器类型（list、tuple、dict、set），有特别的语法，例子如下：

```python
names: list[str] = ["Alice", "Bob"]
point: tuple[int, int] = (10, 20)
student: dict[str, int] = {"age": 20, "grade": 90}
unique_ids: set[int] = {1, 2, 3}
```

换言之，可以使用`[]`来指定容器类型中包含的数据类型。

Python作为一个面向对象语言，我们也可以将返回值类型赋值给其他变量，实现类型别名（数据类型：`typing.TypeAlias`/`typing.GenericAlias`），例子如下：

```python
# 以下代码仅可使用在Python 3.12+
Vector = list[float]  # 类型别名
# 你甚至可以这么写
type Point = tuple[float, float]

def normalize(v: Vector) -> float:
    return sum(x ** 2 for x in v) ** 0.5
```

> ⚠ **Warning**
>
> 在Python 3.11及以前，如果需要使用如上的容器内类型声明，则对于这些容器类型，需要使用从`typing`库中导入对应的类型。如果直接使用Python的内置类型，则会报错。例：
>
> ```python
> # 正确写法
> from typing import TypeAlias, List
> Vector= List[float]
> Point = Tuple[float, float]
> ```
>
> ```python
> # 错误写法
> Vector = list[float]
> type Point = tuple[float, float]
> # 报错: TypeError: 'type' object is not subscriptable
> ```

> 除此之外，我们应当注意：在利用此Warning中所写的正确写法，得到的数据类型为`typing.TypeAlias`；而在Python3.10+中使用上述如`list[str]`的方式来声明类型，则得到的数据类型为`typing.GenericAlias`（泛型）。

除此之外，Python中还有专门用于函数/变量声明类型的类型，包括`Optional`、`Union`、`Callable`、`Literal`，例子分别如下：

```
from typing import Optional # 表示值可以是None或指定类型
# 事实上在不使用Optional的返回值类型声明中，也可以直接返回None(指IDE并不会对此提出警告)
# Optional[X] 指要么返回X类型，要么返回None
def find_user(id: int) -> Optional[str]:
    if id == 1:
        return "Alice"
    else:
        return None  # 允许返回None
# Optional[X] 实则等价于 Union[X, None]，于是其在Python3.10+中等价于 X | None
```

```
from typing import Union    # 表示值可以是多种类型之一
# Union[A, B, ...] 返回类型A或类型B或...
def parse_input(value: Union[str, int]) -> float:
    if isinstance(value, str):
        return float(value)
    else:
        return float(value)
# 对于Union[str, int]，在Python3.10+中有如下等价表达：
str | int # 返回值即Union[str, int]

# 关于Union，在容器的数据类型声明中，也可以使用，如：
list[Union[int, str]]
list[int | str] # Python3.10+中的等价表达，以下均略写为此
dict[str, int | str | bool]
dict[int | str, bool | float]
tuple[int | str, bool, str | list]
set[int, str, bool]
```

```
from typing import Callable # 表示一个函数/可调用的类型
# 虽然对于一个函数使用type函数，得到的返回值为<class 'function'>，但是Python并没有声明function的类型，这也就导致了若你需要声明一个变量其类型为函数，只能使用Callable类型，而不能使用funciton
# Callable[[A, B, ...], X] 指此函数/可调用类型所能接受的参数类型依次为A，B，...，且返回类型为X
def apply_func(func: Callable[[int, int], int], a: int, b: int) -> int:
    return func(a, b)
# 对于Callable[[int, int], int]，在Python3.10+中有如下等价表达：
(int, int) -> int # 返回值即Callable[[int, int], int]
```

```
from typing import Literal  # 表示值可以是某些特定取值之一
# Literal[A, B, ...] 指此变量/函数返回值只能为A或B或...
def move(direction: Literal["left", "right", "up", "down"]) -> None:
    print(f"Moving {direction}")
```

附视频教程参考：

https://www.bilibili.com/video/BV11Z4y1h79y
https://www.bilibili.com/video/BV16Y411K73d

§1.7.3 不定长参数（*args、**kwargs）

不定长函数

在 Python 中，不定长函数允许函数接受任意数量的参数，适用于参数数量不确定的场景。Python 提供了两种方式实现可变参数：

1. `*args`：接收任意数量的位置参数，打包为元组。
2. `**kwargs`：接收任意数量的关键字参数，打包为字典，其中键、值在传入参数时形如`键=值`。

> **Tip**
>
> `args`与`kwargs`只是常用名称。如果你想，那么你可以将其换为其他符合Python变量命名规则的变量，并且将以下示例中的`args`/`kwargs`进行替换即可。

> **Important**
>
> 对于`**kwargs`，在传参时的键名不加引号。但是在函数内调用字典`kwargs`时，其中传参对应的键名以字符串类型存储为字典的键，其对应的值即传入参数时等号后的值。详细使用见下例。

具体语法如下：

```
def 函数(固定参数, ..., *args, **kwargs):
    ...
```

比方说：

```
def f(a, b, c, *args, **kwargs):
    print(a)
    print(b)
    print(c)
    print(type(args), args)
    print(type(kwargs), kwargs)

>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument: 'c'

>>> f(1, 2, 3)
1
2
3
<class 'tuple'> ()
<class 'dict'> {}

>>> f(1, 2, 3, 4, 5, 6, 7)
1
2
3
```

```
<class 'tuple'> (4, 5, 6, 7)
<class 'dict'> {}

>>> f(1, 2, 3, d=1, e=2)
1
2
3
<class 'tuple'> ()
<class 'dict'> {'d': 1, 'e': 2}
```

> **Important**
>
> 在使用不定长参数时，请注意参数顺序，必须按照：
> `普通参数`、`位置参数(*args)`、`关键字参数(**kwargs)`的顺序。
>
> 当然，你可以在没有`*args`时直接使用`**kwargs`，或是在没有`**kwargs`时使用`*args`。

除此之外，对于不定长参数，同样也可以使用类型注释（注意，其指定的类型直接为`args`元组中元素的类型即`kwargs`字典中值的类型，而不是tuple或dict），例如：

```
def f(*args: str, **kwargs: int | str):
    ...
```

容器解包

在传入参数时，你可以选择如上的不定长函数，将参数打包为元组或字典，那么也就可以使用类似的方式，将元组/列表或字典拆开，当作参数传入函数。与上类似的，对应的符号也是`*`与`**`。使用`*`解包序列（如列表/元组）为位置参数，而使用`**`解包字典为关键字参数。示例如下：

```
numbers = [1, 2, 3]
sum_numbers(*numbers)  # 等价于 sum_numbers(1, 2, 3)

config = {"host": "localhost", "port": 8080}
connect_to_db(**config)  # 等价于 connect_to_db(host="localhost", port=8080)
```

§1.7.4 try、except、finally

在Python中，可以使用`try-except-else-finally`系列语句处理异常，基本语法结构为：

```
try:
    # 可能引发异常的代码
    risky_operation()
except ExceptionType1:
    # 异常处理代码1
    handle_error1()
except ExceptionType2:
    # 异常处理代码2
    handle_error2()
... # 其他异常处理
except:
    # 其他异常处理代码
```

```
    handle_error()
else:
    # 无异常时处理
    no_error_operation()
finally:
    # 无论是否发生异常都会执行的代码
    cleanup()
```

完整执行流程如下：

1. 先执行 `try` 块中的代码。
2. 如果发生异常：
    - 匹配 `except` 块并执行对应的处理逻辑。
    - 跳过 `try` 块剩余的代码。
3. 如果没有发生异常：
    - 执行`else`块代码（如果写有`else`块）
4. 无论是否发生异常：
    - 若写有`finally` 块，则`finally` 块始终执行（即使 `try` 或 `except` 中有 `return` 或未处理的异常）。

> **Important**
>
> `else`块代码可选，可有可无。
>
> 无论如何，`except`块与`finally`块至少要有其一，否则会抛出`SyntaxError: expected 'except' or 'finally' block`错误。

通常来说，对于`except`块，可以选择后面跟上特定的错误类型，也可以不跟上错误类型。

> **Important**
>
> 若错误类型与某个`except`后错误类型的匹配，则不会在运行不跟错误类型的`except`块。
>
> 在不跟错误类型的`except`块后面，不能再有跟有特定错误类型的`except`块。 只能跟有`else`块及`finally`块（如果需要）。
>
> 如果没有不跟错误类型的`except`块，则在`try`块中运行出现与所有`except`块后跟错误类型均不匹配的错误后，仍会抛出错误并终止程序。

但是在开发中，仍有需注意的：

1. `finally` 的优先级最高：
    - 即使 `try` 或 `except` 中有 `return`、`break` 或未处理的异常，`finally` 仍会执行。

```
def test():
    try:
        return "try"
    finally:
        print("finally")  # 会先执行

print(test())  # 输出: finally → try
```

2. 避免使用不跟有错误类型的 `except`：
    - 不跟有错误类型的`except`会捕获所有异常（包括 `KeyboardInterrupt` 和 `SystemExit`），可能隐藏严重错误。

```
try:
    risky_call()
except:  # 不推荐！应明确异常类型
    print("发生错误")
```

§1.7.5 for-else与while-else

在Python，`else`语句除了跟在`if`/`elif`语句及`try`/`except`语句之后外，也可以跟在`for`/`while`语句之后。其逻辑与`try`/`except`语句类似：若`for`/`while`循环结束时不是通过`break`语句结束的，则会运行`else`后代码块，否则则不运行。其与`try-else`的设计思想一致，均为正常完成则执行。使用示例如下：

```
numbers = [1, 3, 5, 7]
for num in numbers:
    if num % 2 == 0:
        print("发现偶数")
        break
else:
    print("未发现偶数")  # 输出（因为循环未被 break）

# 等价于
found = False
for num in numbers:
    if num % 2 == 0:
        print("发现偶数")
        found = True
        break
if not found:
    print("未发现偶数")
```

```
max_attempts = 3
attempt = 0

while attempt < max_attempts:
    password = input("输入密码: ")
    if password == "123456":
        print("登录成功")
        break
    attempt += 1
else:
    print("尝试次数过多，账户锁定")  # 仅在未触发 break 时执行

# 等价于
success = False
while attempt < max_attempts:
    password = input("输入密码: ")
    if password == "123456":
```

```
            print("登录成功")
            success = True
            break
        attempt += 1
    if not success:
        print("尝试次数过多，账户锁定")
```

§1.7.6 生成器（Generator）、迭代器（Iterator）与Range

Generator、Iterator、Range都是Python中用于处理可迭代数据的工具，但它们在实现方式、内存效率和适用场景上有显著区别，具体如下：

特性	Generator Object	Range Object	Iterator
创建方式	`yield` 或生成器表达式	`range()` 函数	实现 `__next__()`
内存效率	高（惰性生成）	高（存储起止步长）	依赖实现
是否可重置	否（一次性）	是（可重复迭代）	通常否
支持操作	`send()` / `throw()` 等、迭代	切片、索引、`in` 操作、迭代	仅基础迭代
典型用途	流式数据处理	固定范围的整数序列	自定义迭代逻辑

关于这些内容，此处不作单独介绍。事实上，其中细节在日常开发中并不常用。如需了解，可以参见：

6.2.8. 生成器表达式 | Python 3.13.4 文档：https://docs.python.org/zh-cn/3.13/reference/expressions.html#generator-expressions

6.2.9. yield 表达式 | Python 3.13.4 文档：https://docs.python.org/zh-cn/3.13/reference/expressions.html#yield-expressions

迭代器类型 | Python 3.13.4 文档：https://docs.python.org/zh-cn/3.13/library/stdtypes.html#iterator-types

§1.7.7 Python中类的魔法函数（magic method / special method）

魔法函数是Python的一种高级语法，允许你在类中自定义函数（函数名格式一般为`__xx__`），并绑定到类的特殊方法中。比如在类A中自定义`__str__`函数，则在调用`str(<A object>)`时，会自动调用`__str__`函数，并返回相应的结果。

此处暂不对于具体魔法函数进行文字版介绍。如有需要，请参考：

Python文档：3.3. 特殊方法名称 | Python 3.13.4 文档：https://docs.python.org/zh-cn/3/reference/datamodel.html#special-method-names

详细介绍视频：合集·Python魔术方法大全 | 码农高天 | 哔哩哔哩：https://space.bilibili.com/245645656/lists/1023741

当然，这里也期待后续的社员将此节完善。

§1.7.8 eval与exec

Python作为一个解释型语言，除了直接利用Python解释器运行Python源码文件外，你也可以利用`eval`与`exec`函数在Python代码中运行以str格式存储的Python代码。当然，`eval`与`exec`之间也有一定的区别。

eval与exec简单运用

`eval`函数用于计算单个表达式的值（包括计算式、函数等）并返回结果，例如：

```
x = 10
x_result = eval('x * 2 + 5')    # 返回25

l = [3, 4, 7, 3]
l_result = eval('sorted(l)')    # 返回[3, 3, 4, 7]
```

`exec`函数用于执行包含Python代码的字符串或代码对象，例如：

```
code = """
def greet(name):
    print(f"Hello, {name}!")
greet("Alice")
"""
exec(code)  # 输出: Hello, Alice!
```

而`eval`与`exec`存在一定的运行与语法差异，如下：

特性	eval	exec
输入内容	只能接受单个表达式	可以接受代码块、语句、函数/类定义等
返回值	返回表达式的计算结果	总是返回 `None`，如需获取结果必须通过变量赋值
分号处理	不能使用分号分隔多个表达式	可以用分号分隔多个语句
多行代码	不支持	支持多行代码块
赋值语句	不能直接包含赋值语句	可以包含赋值语句
函数/类定义	不能定义函数或类	可以定义函数和类
用途	计算简单表达式	执行复杂代码
安全性	同样危险	同样危险

> ⚠ **Warning**
>
> 避免以下的错误使用

```
# eval() 不能执行语句
eval("a = 10")  # SyntaxError

# exec() 不能直接获取表达式结果
x = exec("3 + 4")  # x 将是 None，不是 7
```

eval与exec的参数与安全处理

`eval`与`exec`的完整语法如下：

```
eval(__source: str, __globals: dict[str, Any]=None, __locals: dict[str, Any]=None)
exec(__source: str, __globals: dict[str, Any]=None, __locals: dict[str, Any]=None, ...)
```

其中`__globals`与`__locals`作为可选参数，对于`eval`与`exec`中作用基本相同，用于指定对应的命名空间（字典形式，类似`**kwargs`在函数中的值），有利于提高`eval`与`exec`的安全性，一定程度上避免通过这两个函数进行注入攻击。具体含义如下：

参数	作用	默认值
`__globals`	指定全局命名空间，决定代码可以访问哪些全局变量和函数	默认为当前全局命名空间`globals()`
`__local`	指定局部命名空间，决定代码可以访问哪些局部变量	默认为当前局部命名空间`locals()`

其具体运用效果分别如下：

- 只提供`__globals`（`__locals`使用`__globals`的副本）

```
my_globals = {'a': 1, 'b': 2}
print(eval('a + b', my_globals))  # 输出 3

# 无法访问真实全局变量
x = 10
print(eval('x', my_globals))  # NameError: name 'x' is not defined
```

- 同时提供`__globals`和`__locals`

```
my_globals = {'a': 1}
my_locals = {'b': 2}
print(eval('a + b', my_globals, my_locals))  # 输出 3
```

> **Important**
>
> 虽然在以上的语法介绍中，为每个参数提供了一个名称，但事实上`eval`和`exec`在定义时均定义为：
>
> ```
> def eval(*args): ...
> def exec(*args, **kwargs): ... # **kwargs是用于传入闭包变量的，通常不被使用
> ```

所以实际上`eval`和`exec`并不能通过关键词参数传入命名空间，也就导致了不能仅传入局部命名空间而不传入全局命名空间。

> **Tip**
>
> 传入命名空间具有写回的特性，即会对传入的字典进行修改（通常发生在`exec`函数中，因为`eval`函数通常不会修改传入字典），例如：
>
> ```
> my_locals = {}
> exec("x = 10", {}, my_locals)
> print(my_locals)  # 输出 {'x': 10}
> ```
>
> 而在对待变量（读与写）时，与Python代码块语法类似：
>
> - 如果变量在`__locals`中存在，则直接读写`__locals`对应变量
> - 否则，如果变量在`__globals`中存在，则读写`__globals`对应变量
> - （在变量赋值时），如果变量同时不存在与`__locals`和`__globals`，则会在`__locals`中新建变量
>
> 但是请注意，正如上文所说，在只提供`__globals`时，`__locals`会使用`__globals`的副本。这也就导致了只提供`__globals`时，对变量读写会一律作用与`__globals`

> **⚠ Warning**
>
> Python作为面向对象语言，函数与类型都是存储于变量中的。这也就导致了，在传入自定义命名空间时，无法使用未包含在命名空间中的自定义函数/类型。
>
> 若全局命名空间中不存在键`__builtins__`，则会自动运行函数时，将Python的`__builtins__`传入自定义全局命名空间中，并在运行结束后保留，例如：
>
> ```
> c = {}
> exec('', c)
> print(c)
> ```
>
> 就会输出`{'__builtins__': {...(过长略去)}}`
>
> 而这也导致，如果你传入的全局命名空间中存在自定义的`__builtins__`，则未被包含的默认函数/类也无法被使用，如：
>
> ```
> eval('print(1)', {'__builtins__': {}})  # 输出: NameError: name 'print' is not defined
> ```

§1.7.9 断言（assert）

断言（assert）是Python中的一个调试辅助工具，用于在代码中设置检查点，确保程序在某个时刻满足特定条件。如果断言条件不满足，程序会抛出`AssertionError`异常。基本语法如下：

```
assert condition(, message)
```

其中`condition`指要测试的条件表达式，`message`（可选）指当断言失败时要显示的错误消息（可以是任意类型，对其使用`str()`函数即得到输出的消息）

使用示例如下：

```
# 简单断言
x = 5
assert x == 5   # 不会引发异常
assert x == 3   # 引发 AssertionError

# 带错误消息的断言
assert x % 2 == 0, "x必须是偶数" # 错误消息: AssertionError: x必须是偶数

# 函数参数检查
def divide(a, b):
    assert b != 0, "除数不能为零"
    return a / b
```

> **Important**
>
> 在运行Python时，添加`-O`（大写字母O）选项可以禁用所有断言，即：
>
> ```
> python -O script.py
> ```

在使用断言时，应当：

- 使用断言检查"不应该发生"的情况
- 提供清晰、有用的错误消息
- 不要用断言替代异常处理
- 断言条件应该没有副作用（不改变程序状态）

§1.7.10 匿名函数（lambda）

lambda 函数是 Python 中的匿名函数（没有名称的函数），使用 `lambda` 关键字创建。它们适用于需要小函数但不想正式定义函数的场景，基本语法如下：

```
(f =) lambda arguments: expression
```

- `f`：（可选）将函数赋值给的变量，后可直接当作函数使用
- `lambda`：关键字，表示开始一个 lambda 函数
- `arguments`：函数的参数，可以是一个或多个，用逗号分隔
- `expression`：单个表达式，是函数的返回值（不能包含语句）

对于上述表达，可以等同于：

```
def f(arguments):
    return expression
```

> **Tip**
>
> 对于全局变量的读写，与直接声明的函数调用相同，例：

```
a = 0
f1 = lambda:(a:=1)
print(f1(), a)
# 输出: 1 0

def f2():
return (a:=1)
print(f2(), a)
```

而与正常的函数声明相同，`lambda`也支持多种不同的参数，例如下

```
# 多参数 lambda
add = lambda a, b: a + b
print(add(3, 5))  # 输出: 8

# 无参数 lambda
get_answer = lambda: 42
print(get_answer())  # 输出: 42

# 默认参数
power = lambda x, y=2: x ** y
print(power(3))     # 输出: 9 (使用默认y=2)
print(power(3, 3))  # 输出: 27

# 可变参数
sum_all = lambda *args: sum(args)
print(sum_all(1, 2, 3, 4))  # 输出: 10
person_info = lambda **kwargs: "; ".join(f"{k}:{v}" for k, v in kwargs.items())
print(person_info(name="Alice", age=25, city="NY")) # 输出: name:Alice; age:25; city:NY

# 类型注释
add: Callable[[int, int], int] = lambda x, y: x + y
# 由于冒号语义的冲突，所以单独的lambda并不支持参数及返回值类型标注
```

> **Tip**
>
> 你也可以直接这么使用：
>
> ```
> (lambda ...: ...)(...)
> ```

通常来说，`lambda`被用于需要传入函数为参数的函数且无需创建单独的函数的时候，如：

```
# 与 map() 配合
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))

# 与 filter() 配合
evens = list(filter(lambda x: x % 2 == 0, numbers))

# 与 sorted() 配合
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

```
pairs.sort(key=lambda pair: pair[1])

data = [{'name': 'John', 'age': 45}, {'name': 'Diana', 'age': 35}, {'name': 'Aaron', 'age': 40}]
sorted_by_name = sorted(data, key=lambda x: x['name'])
sorted_by_age = sorted(data, key=lambda x: x['age'])

# 在 GUI 编程中作为回调函数(如Tkinter)
button = Button(text="Click", command=lambda: print("Button clicked"))
```

> **Important**
>
> `lambda`函数只能包含一个表达式，不能包含复杂的逻辑（分号等）或多条语句，也不包含`return`，也不能包含`if`、`for`、`while`等代码块，但是可以使用“单行if-else语句”，也可以以包含“单行for语句”的列表/元组/生成器作返回值。

> **Tip**
>
> 复杂的`lambda`函数会降低代码可读性，且在调试时，`lambda`函数错误堆栈跟踪中显示为`<lambda>`，调试时不如命名函数方便。在逻辑较复杂时，应该使用常规的 def 函数。
>
> 但是`lambda`函数和普通函数在性能上没有显著差异，选择使用哪种形式应基于可读性和使用场景。

§1.8 图形交互初步

§1.8.0 Tkinter 简介

Tkinter是Python的标准GUI工具包，基于Tk GUI工具包构建。它是Python内置库，无需额外安装。而作为Python内置库，其使用也颇具面向对象特色。但是作为入门，Tkinter中的面向对象语法此处不作介绍。仅介绍一般普通用法。

§1.8.1 Tkinter窗口搭建

语法规范

```
import tkinter as tk

# 创建主窗口
root = tk.Tk()  # 创建Tkinter主窗口对象

# 窗口配置方法:
root.title("窗口标题")       # 设置窗口标题
root.geometry("宽x高")      # 设置窗口初始大小，如"400x300"，单位为像素，受系统dpi影响，下同
root.resizable(宽可调，高可调)  # 设置窗口是否可调整大小，True/False
root.minsize(最小宽，最小高)  # 设置窗口最小尺寸
root.maxsize(最大宽，最大高)  # 设置窗口最大尺寸

# 主事件循环
root.mainloop()  # 启动GUI应用程序，此代码可看作一个while True: ...语句，程序会被暂停控制在此行代码
```

示例

```
import tkinter as tk

root = tk.Tk()
root.title("teaGfx DirectX Release")
root.geometry("1920x1080")
root.resizable(True, False)  # 宽度可调，高度固定
root.minsize(1920, 1080)

root.mainloop()
```

§1.8.2 基本组件详解

标签 (Label)

语法规范

```
label = tk.Label(
    master,          # 父容器
    text="文本",     # 显示的文本
    font=("字体", 大小, "样式"),  # 字体设置
    fg="前景色",     # 文字颜色
    bg="背景色",     # 背景颜色
    width=宽度,      # 标签宽度(字符单位，下同)
    height=高度,     # 标签高度(字符单位)
    anchor=对齐位置, # 文本对齐(n, ne, e, se, s, sw, w, nw, center)，其含义即为东南西北方位，符合上北下南左西右东
    relief=边框样式  # 边框样式(flat, groove, raised, ridge, solid, sunken)，具体图例可上网查阅
)
label.pack()  # 或其他布局方法
```

> [!Tip]
>
> 关于像素`px`与字符单位`ex`并不等同。一般而言，有1ex=16px，但是在有的时候会出现并不对应的情况。

示例

```
label = tk.Label(
    root,
    text="欢迎使用Tkinter",
    font=("微软雅黑", 12, "bold"),
    fg="blue",
    bg="#f0f0f0",
    width=20,
    height=2,
    anchor="center",
    relief="ridge"
)
label.pack(pady=10)
```

按钮 (Button)

语法规范

```
button = tk.Button(
    master,
    text="按钮文本",
    command=函数名,  # 点击时调用的函数(不带括号, 为function对象, 此处可适当使用lambda匿名函数)
    width=宽度,
    height=高度,
    fg="文字颜色",
    bg="背景颜色",
    activeforeground="按下时文字颜色",
    activebackground="按下时背景颜色",
    state="状态"  # normal(默认), active, disabled
)
button.pack()
```

示例

```
def button_click():
    print("按钮被点击了")

button = tk.Button(
    root,
    text="点击我",
    command=button_click,
    width=15,
    height=2,
    fg="white",
    bg="#4CAF50",
    activeforeground="black",
    activebackground="#8BC34A",
    state="normal"
)
button.pack(pady=10)
```

输入框 (Entry)

语法规范

```
entry = tk.Entry(
    master,
    width=宽度,  # 字符宽度
    font=字体,
    fg="文字颜色",
    bg="背景颜色",
    show="替代字符",  # 如密码框用"*"
    state="状态",  # normal, disabled
    relief=边框样式
)
entry.pack()
```

常用方法

- `entry.get()` - 获取输入内容
- `entry.delete(起始, 结束)` - 删除内容
- `entry.insert(位置, 文本)` - 插入文本

示例

```
entry = tk.Entry(
    root,
    width=30,
    font=("Arial", 12),
    fg="black",
    bg="white",
    show=None,
    state="normal",
    relief="sunken"
)
entry.pack(pady=10)

# 获取内容示例
def get_entry():
    content = entry.get()
    print("输入内容:", content)

tk.Button(root, text="获取输入", command=get_entry).pack()
```

文本框 (Text)

语法规范

```
text = tk.Text(
    master,
    width=宽度,  # 字符宽度
    height=高度,  # 行数
    font=字体,
    fg="文字颜色",
    bg="背景颜色",
    wrap="换行模式",  # char, word, none
    padx=水平内边距,
    pady=垂直内边距
)
text.pack()
```

常用方法

- `text.get(起始, 结束)` - 获取文本
- `text.insert(位置, 文本)` - 插入文本
- `text.delete(起始, 结束)` - 删除文本
- `text.tag_add(标签名, 起始, 结束)` - 添加文本标签
- `text.tag_config(标签名, 样式)` - 配置标签样式

示例

```
text = tk.Text(
    root,
    width=40,
    height=10,
    font=("Consolas", 11),
    fg="black",
    bg="#f8f8f8",
    wrap="word",
    padx=5,
    pady=5
)
text.pack(pady=10)

# 插入示例文本
text.insert("1.0", "这是一个多行文本框\n")
text.insert("end", "可以输入和编辑多行文本内容\n")

# 添加样式标签
text.tag_add("highlight", "1.0", "1.5")
text.tag_config("highlight", background="yellow", foreground="red")

# 获取内容示例
def get_text():
    content = text.get("1.0", "end-1c")  # end-1c去掉最后的换行符
    print("文本框内容:", content)

tk.Button(root, text="获取文本框内容", command=get_text).pack()
```

单选按钮 (Radiobutton)

语法规范

```
var = tk.StringVar()  # 或 IntVar()

radio = tk.Radiobutton(
    master,
    text="显示文本",
    variable=共享变量,
    value=选项值,
    command=回调函数,
    font=字体,
    fg="文字颜色",
    bg="背景颜色"
)
radio.pack()
```

示例

```
radio_var = tk.StringVar(value="Python")

def radio_selected():
    print("选择了:", radio_var.get())

options = [("Python", "Python"), ("Java", "Java"), ("C++", "C++"), ("JavaScript", "JS")]

for text, value in options:
    tk.Radiobutton(
        root,
        text=text,
        variable=radio_var,
        value=value,
        command=radio_selected,
        font=("Arial", 10),
        fg="navy",
        bg="#f0f0f0"
    ).pack(anchor="w", padx=20)
```

复选框 (Checkbutton)

语法规范

```
var = tk.IntVar()  # 0未选中，1选中

check = tk.Checkbutton(
    master,
    text="显示文本",
    variable=变量,
    onvalue=选中值,
    offvalue=未选中值,
    command=回调函数,
```

```
    font=字体,
    fg="文字颜色",
    bg="背景颜色"
)
check.pack()
```

示例

```
check_var1 = tk.IntVar(value=1)  # 默认选中
check_var2 = tk.IntVar(value=0)  # 默认未选中

def check_changed():
    print(f"选项1: {'选中' if check_var1.get() else '未选中'}, "
          f"选项2: {'选中' if check_var2.get() else '未选中'}")

tk.Checkbutton(
    root,
    text="记住密码",
    variable=check_var1,
    onvalue=1,
    offvalue=0,
    command=check_changed,
    font=("Arial", 10),
    fg="black",
    bg="#f0f0f0"
).pack(anchor="w", padx=20)

tk.Checkbutton(
    root,
    text="自动登录",
    variable=check_var2,
    onvalue=1,
    offvalue=0,
    command=check_changed,
    font=("Arial", 10),
    fg="black",
    bg="#f0f0f0"
).pack(anchor="w", padx=20)
```

列表框 (Listbox)

语法规范

```
listbox = tk.Listbox(
    master,
    width=宽度,
    height=高度,
    font=字体,
    fg="文字颜色",
    bg="背景颜色",
    selectmode="选择模式",  # single, browse, multiple, extended
    activestyle="激活样式"  # none, dotbox, underline
)
listbox.pack()
```

常用方法：

- `listbox.insert(位置, 项目)` - 添加项目
- `listbox.delete(起始, 结束)` - 删除项目
- `listbox.get(起始, 结束)` - 获取项目
- `listbox.curselection()` - 获取选中项的索引

示例：

```
listbox = tk.Listbox(
    root,
    width=30,
    height=5,
    font=("Arial", 10),
    fg="black",
    bg="white",
    selectmode="browse",
    activestyle="dotbox"
)
listbox.pack(pady=10)

# 添加项目
for item in ["苹果", "香蕉", "橙子", "葡萄", "西瓜", "芒果", "草莓"]:
    listbox.insert("end", item)

# 添加滚动条
scrollbar = tk.Scrollbar(root)
scrollbar.pack(side="right", fill="y")
listbox.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=listbox.yview)

# 获取选中项
def get_selected():
    selected = listbox.curselection()
    if selected:
        print("选中的是:", listbox.get(selected[0]))

tk.Button(root, text="获取选中项", command=get_selected).pack()
```

下拉菜单 (OptionMenu)

语法规范

```
var = tk.StringVar()
var.set(默认值)

option_menu = tk.OptionMenu(
    master,
    var,
    *选项列表,
    command=回调函数
)
option_menu.pack()
```

示例

```
options = ["北京", "上海", "广州", "深圳", "杭州", "成都"]
city_var = tk.StringVar(value=options[0])

def city_changed(*args):
    print("选择的城市:", city_var.get())

option_menu = tk.OptionMenu(
    root,
    city_var,
    *options,
    command=city_changed
)
option_menu.config(
    font=("Arial", 10),
    fg="black",
    bg="white",
    width=15,
    height=1
)
option_menu.pack(pady=10)
```

滚动条 (Scrollbar)

语法规范

```
scrollbar = tk.Scrollbar(
    master,
    orient="方向",  # vertical或horizontal
    command=滚动控制函数
)
scrollbar.pack(side="位置", fill="填充方向")

# 关联滚动条和组件
widget.config(yscrollcommand=scrollbar.set)
```

示例

```
text = tk.Text(
    root,
    width=40,
    height=10,
    font=("Consolas", 11),
    wrap="word"
)

scrollbar = tk.Scrollbar(
    root,
    orient="vertical",
    command=text.yview
)
scrollbar.pack(side="right", fill="y")

text.pack(side="left", fill="both", expand=True)
text.config(yscrollcommand=scrollbar.set)

# 插入示例文本
for i in range(50):
    text.insert("end", f"这是第 {i+1} 行文本\n")
```

框架 (Frame)

语法规范

```
frame = tk.Frame(
    master,
    width=宽度,
    height=高度,
    bg="背景颜色",
    bd=边框宽度,
    relief=边框样式,  # flat, groove, raised, ridge, solid, sunken
    padx=水平内边距,
    pady=垂直内边距
)
frame.pack()
```

示例

```
frame = tk.Frame(
    root,
    width=300,
    height=200,
    bg="#f0f0f0",
    bd=2,
    relief="groove",
    padx=10,
    pady=10
)
```

```
frame.pack(pady=10, fill="both", expand=True)

# 在框架内添加组件
tk.Label(frame, text="框架内的标签", bg="#f0f0f0").pack()
tk.Button(frame, text="框架内的按钮").pack()
```

消息框 (Messagebox)

语法规范

```
from tkinter import messagebox

# 信息框
messagebox.showinfo("标题", "消息内容")

# 警告框
messagebox.showwarning("标题", "警告内容")

# 错误框
messagebox.showerror("标题", "错误内容")

# 询问框(返回"yes"或"no")
result = messagebox.askyesno("标题", "问题内容")

# 确认框(返回True或False)
result = messagebox.askokcancel("标题", "确认内容")

# 重试取消框(返回True或False)
result = messagebox.askretrycancel("标题", "重试内容")
```

示例

```
from tkinter import messagebox

def show_dialogs():
    messagebox.showinfo("信息", "这是一个信息提示框")
    messagebox.showwarning("警告", "这是一个警告框")
    messagebox.showerror("错误", "这是一个错误框")

    if messagebox.askyesno("问题", "你确定要继续吗?"):
        print("用户选择了是")
    else:
        print("用户选择了否")

    if messagebox.askokcancel("确认", "真的要删除吗?"):
        print("用户确认了操作")
    else:
        print("用户取消了操作")

tk.Button(root, text="显示各种消息框", command=show_dialogs).pack(pady=10)
```

§1.8.3 布局管理详解

pack布局

语法规范

```
widget.pack(
    side="位置",  # top, bottom, left, right
    fill="填充",  # none, x, y, both
    expand=是否扩展,  # True/False
    padx=水平外边距,
    pady=垂直外边距,
    ipadx=水平内边距,
    ipady=垂直内边距,
    anchor="锚点位置"  # n, ne, e, se, s, sw, w, nw, center
)
```

示例：

```
tk.Label(root, text="顶部", bg="red").pack(side="top", fill="x")
tk.Label(root, text="底部", bg="green").pack(side="bottom", fill="x")
tk.Label(root, text="左边", bg="blue").pack(side="left", fill="y", expand=True)
tk.Label(root, text="右边", bg="yellow").pack(side="right", fill="y", expand=True)
```

grid布局

语法规范

```
widget.grid(
    row=行号,  # 从0开始
    column=列号,  # 从0开始
    rowspan=跨越行数,
    columnspan=跨越列数,
    padx=水平外边距,
    pady=垂直外边距,
    ipadx=水平内边距,
    ipady=垂直内边距,
    sticky="对齐方式"  # n, s, e, w, ne, nw, se, sw的组合
)
```

示例：

```
for i in range(3):
    for j in range(3):
        tk.Label(
            root,
            text=f"行{i}列{j}",
            borderwidth=1,
            relief="solid",
            width=10,
            height=3
```

```
        ).grid(
            row=i,
            column=j,
            padx=5,
            pady=5,
            sticky="nsew"
        )

# 配置行列权重
for i in range(3):
    root.grid_rowconfigure(i, weight=1)
    root.grid_columnconfigure(i, weight=1)
```

place布局

语法规范

```
widget.place(
    x=绝对x坐标,
    y=绝对y坐标,
    relx=相对x位置(0-1),
    rely=相对y位置(0-1),
    anchor="锚点位置",  # n, ne, e, se, s, sw, w, nw, center
    width=宽度,
    height=高度
)
```

示例:

```
tk.Label(root, text="绝对位置", bg="red").place(x=50, y=50, width=100, height=50)
tk.Label(root, text="相对位置", bg="blue").place(relx=0.5, rely=0.5, anchor="center")
tk.Label(root, text="右下角", bg="green").place(relx=1.0, rely=1.0, anchor="se")
```

> **Tip**
>
> pack、grid、place布局的优缺点与推荐使用环境

特性	pack 布局	grid 布局	place 布局
基本原理	按照添加顺序排列组件	基于行/列的网格系统	绝对或相对位置定位
优点	- 简单易用 - 自动调整大小 - 适合简单布局	- 灵活强大的网格系统 - 行列对齐精确 - 适合复杂布局	- 像素级精确控制 - 可重叠组件 - 支持相对定位
缺点	- 复杂布局难以实现 - 控制精度有限 - 不能混合使用	- 代码量稍多 - 网格需要预先规划	- 不随窗口自动调整 - 维护困难 - 跨平台显示可能不一致
推荐使用场景	- 简单垂直/水平排列 - 工具栏/状态栏 - 快速原型开发	- 表单类应用 - 复杂界面布局 - 需要行列对齐的界面	- 需要精确控制位置 - 自定义绘图组件 - 游戏界面元素

特性	pack 布局	grid 布局	place 布局
混合使用	可在不同容器中与grid混合使用	可在不同容器中与pack混合使用	尽量避免与其他布局混合使用
代码示例	`widget.pack(side=LEFT, fill=Y)`	`widget.grid(row=0, column=1)`	`widget.place(x=50, y=30)`

使用建议：

1. 优先选择`grid`：适合大多数应用场景，特别是需要精确对齐的界面
2. 简单布局用`pack`：当只需要简单堆叠组件时更便捷
3. 谨慎使用`place`：只在需要绝对定位或特殊效果时使用
4. 混合布局策略：在不同容器中可以使用不同布局管理器
5. 注意：在同一个容器内不要混用pack和grid，这会导致布局问题。但可以在不同容器中使用不同的布局管理器。

§1.9 第三方库介绍及常用库推荐

§1.9.0 第三方库简介

第三方库，或第三方模块，又称轮子，可以供其他开发者使用已经开发好了的函数/方法。通常来说，第三方库托管在pypi仓库上，可以方便地通过pip进行安装和管理。而丰富的第三方库、其便捷的安装，以及大部分第三方库均可跨平台使用，给Python在各种编程语言的竞争中赢得了极大的优势。

§1.9.1 第三方库的安装与管理

此处仅介绍在Windows的默认环境下的安装。对于虚拟环境venv以及MacOS、Linux等环境下操作仅略有不同，此处不作介绍。

此处默认在安装Python过程中，已勾选"Add python.exe to PATH"。

> **Tip**
>
> 通常来说，以下所有的`pip`命令都可以使用`python -m pip`替代。

关于pip命令，首先可以使用

```
pip --version
```

来验证pip是否能正常使用，并且了解pip的版本。

> **Tip**
>
> 通常来说，在使用任意pip语句后，如果当前pip不是最新版，都会给出例如以下的提示（较新版，老板pip的更新提示这里找不到了）：
>
> ```
> [notice] A new release of pip is available: 24.0 -> 25.1.1
> [notice] To update, run: python.exe -m pip install --upgrade pip
> ```
>
> 关于是否更新，这里给出优缺点：

> - 优点：更新的pip修复了bug，并且会有更现代化的交互样式。
> - 缺点：更新过程中如果出现问题，可能会难以挽回，需要重新安装Python。

如果你的英语和理解能力足够，建议使用`pip -h`的命令，直接阅读官方使用的帮助文档。当然，也可以参考以下教程。

pip命令基本格式为：

```
pip <command> [options]
```

其中常用的`<command>`包括：

- install：安装库
- uninstall：卸载库
- list：查看已安装的所有库
- show：查看具体库的信息
- help：查看帮助

其中list和help通常不加参数，直接使用即可；install、uninstall和show后面跟上包名进行安装/卸载/查看信息，例如：

```
pip install requests
pip show requests
pip uninstall requests
```

当然，这三个命令后边也可以跟上多个包名，并用空格分隔。

下面给出install的特别语法：

```
: 安装特定版本
pip install 包名==版本号
: 安装特定范围内的版本
pip install "包名>最低版本,<最高版本"
: 升级包
pip install --upgrade 包名
: 使用镜像源
pip install -i 镜像源链接 包名
```

> **Tip**
>
> 在国内直接连接pypi服务器可能较慢，所以在国内使用pip通常会配置镜像源。首先给出一些常用的国内镜像源链接：
>
> - 清华大学：https://pypi.tuna.tsinghua.edu.cn/simple
> - 阿里云：http://mirrors.aliyun.com/pypi/simple/
> - 中国科技大学：https://pypi.mirrors.ustc.edu.cn/simple/
> - 豆瓣：http://pypi.douban.com/simple/

> 其次除了上述通过`-i`参数临时配置镜像网站之外，还可以通过更改配置文件以添加全局镜像网站，使用如下命令即可：
>
> ```
> pip config set global.index-url 镜像源链接
> ```

然后介绍一下利用requirements.txt进行库的安装与迁移。我们可以利用pip生成包含当前所有安装的包的requirements.txt，然后利用此requirements.txt在其他环境下快速配置相同的第三方库运行环境。具体命令如下：

```
: 从requirements文件安装
pip install -r requirements.txt
: 生成requirements文件
pip freeze > requirements.txt
```

§1.9.2 少数第三方库推荐

Pygame

Pygame 是一个用于开发2D游戏的Python库，基于SDL（Simple DirectMedia Layer）库，提供图形渲染、声音播放、事件处理等功能。

- 优点：
  - 简单易用，适合游戏开发初学者。
  - 跨平台支持（Windows、Linux、macOS）。
  - 提供丰富的游戏开发接口（精灵、碰撞检测、键盘/鼠标事件等）。
- 缺点：
  - 性能较低，不适合3D或高性能游戏。
  - 官方更新较慢，社区支持不如商业引擎（如Unity、Unreal Engine）。

PyInstaller

PyInstaller 用于将Python脚本打包成独立的可执行文件（支持Windows、Linux、macOS），无需安装Python环境即可运行。

- 优点：
  - 支持跨平台打包。
  - 自动处理依赖项，无需手动配置。
  - 支持单文件或目录打包。
- 缺点：
  - 打包后的文件体积较大（包含Python解释器和依赖库）。
  - 部分特殊库可能需要手动配置（如PyQt、OpenCV）。

NumPy

NumPy 是Python的科学计算基础库，提供多维数组（ndarray）、线性代数、傅里叶变换等功能。

- 优点：
  - 高性能（底层用C实现）。
  - 丰富的数学运算支持。
  - 是许多科学计算库（如Pandas、SciPy）的基础。
- 缺点：
  - 语法与标准Python有差异，学习成本较高。
  - 仅适用于数值计算，不支持符号运算。

SymPy

SymPy 是一个符号计算库，支持代数、微积分、离散数学等符号运算。

- 优点：
  - 纯Python实现，易于扩展。
  - 支持符号表达式化简、求导、积分等，功能广泛。
- 缺点：
  - 计算速度较慢（相比数值计算库如NumPy）。
  - 功能局限于符号计算，不适用于数值优化。
  - 算法存在部分缺陷，可能出现死循环等情况。

Requests

Requests 是一个HTTP请求库，用于发送GET、POST等网络请求，处理响应数据。

- 优点：
  - 简单易用，API设计优雅。
  - 支持会话（Session）、Cookie、HTTPS等高级功能。
- 缺点：
  - 不支持异步请求（需配合aiohttp等库）。
  - 功能较基础，复杂爬虫需结合其他库（如BeautifulSoup）。

BeautifulSoup

BeautifulSoup 是一个HTML/XML解析库，用于从网页中提取数据。

- 优点：
  - 解析方式灵活（支持lxml、html.parser等后端）。
  - 提供简单的DOM树遍历方法。
- 缺点：
  - 依赖外部解析器（如lxml）。
  - 性能不如Scrapy等专业爬虫框架。

§2 简单算法介绍

§2.0 算法简介

§2.0.0 算法概述

算法，顾名思义，即计算的方法。算法通常用于解决特定的计算任务，但与可以直接在计算机上运行的程序不同，算法使用数学化的描述，更加侧重于思想，可以被看作抽象的程序。同一个算法可以有许多种不同的实现方式，两个不同的程序里也可能使用了同一种算法。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间，空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

本章介绍一些基础算法。当一个计算任务足够通用，在各种问题中广泛出现时，解决该任务的算法就会被更多的算法所调用，拥有相当的重要性和基础性。同时，越通用的算法，通常越是简洁，越近似于思想而非程序。本章中介绍的算法具备足够的优美及趣味性，在之后的进阶内容中也常常会出现。

§2.0.1 复杂度

操作数

同一个算法在不同的计算机上运行的速度会有一定的差别，并且实际运行速度难以在理论上进行计算，实际去测量又比较麻烦，所以我们通常考虑的不是算法运行的实际用时，而是算法运行所需要进行的基本操作的数量。

在普通的计算机上，加减乘除、访问变量（基本数据类型的变量，下同）、给变量赋值等都可以看作基本操作。

对基本操作的计数或是估测可以作为评判算法用时的指标。

例如，以下程序算做a次操作数

```
a = int(input())
for i in range(a):
    print('快来玩Arcaea')
```

时间复杂度

衡量一个算法的快慢，一定要考虑数据规模的大小。所谓数据规模，一般指输入的数字个数、输入中给出的图的点数与边数等等。一般来说，数据规模越大，算法的用时就越长。而在算法竞赛中，我们衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即**时间复杂度**。

考虑用时随数据规模变化的趋势的主要原因有以下几点：

现代计算机每秒可以处理数亿乃至更多次基本运算，因此我们处理的数据规模通常很大。如果算法 A 在规模为 n 的数据上用时为 $100n$ 而算法 B 在规模为 n 的数据上用时为 n^2，在数据规模小于 100 时算法 B 用时更短，但在一秒钟内算法 A 可以处理数百万规模的数据，而算法 B 只能处理数万规模的数据。在允许算法执行时间更久时，时间复杂度对可处理数据规模的影响就会更加明显，远大于同一数据规模下用时的影响。

我们采用基本操作数来表示算法的用时，而不同的基本操作实际用时是不同的，例如加减法的用时远小于除法的用时。计算时间复杂度而忽略不同基本操作之间的区别以及一次基本操作与十次基本操作之间的区别，可以消除基本操作间用时不同的影响。

所谓**用时随数据规模而增长的趋势**是一个模糊的概念，我们需要借助下文所介绍的 渐进符号 来形式化地表示时间复杂度。

渐进符号是函数的阶的规范描述。简单来说，渐进符号忽略了一个函数中增长较慢的部分以及各项的系数（在时间复杂度相关分析中，系数一般被称作**常数**），而保留了可以用来表明该函数增长趋势的重要部分。

常见的渐进符号有Θ，O，Ω，描述复杂度常使用O符号，故此处介绍O符号。读者可自行参阅资料了解其他符号。

> **Important**
>
> O的数学定义：若$f(n)=O(g(n))$, 则存在c，n_0，使得任意$n \geq n_o, 0 \leq f(n) \leq c \cdot g(n)$。其表示函数的渐进下界。研究时间复杂度时通常会使用 O 符号，因为我们关注的通常是程序用时的上界，而不关心其用时的下界。

使用O记号，我们常常可以仅仅通过检查算法的总体结构来描述算法的运行时间，以插入排序为例，双重嵌套循环结构最坏情况运行为$O(n^2)$，也就是说上界为$O(n^2)$。

以下举几种常见的复杂度

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

$O(1)$： 基本执行操作次数是常量
$O(n)$：一般出现在只有一个循环的情况下
$O(n^2)$：一般出现在嵌套循环中
$O(\log n)$：一般出现在二分查找中

空间复杂度

类似地，算法所使用的空间随输入规模变化的趋势可以用**空间复杂度**来衡量。

§2.1 枚举算法

§2.1.0 简介

枚举是基于已有知识来猜测答案的一种问题求解策略.

的思想是不断地猜测，从可能的集合中一一尝试，然后再判断题目的条件是否成立。

§2.1.1 复杂度

$O(n)$

§2.1.2 要点

给出解空间

建立简洁的数学模型。

枚举的时候要想清楚：可能的情况是什么？要枚举哪些要素？

减少枚举的空间

枚举的范围是什么？是所有的内容都需要枚举吗？

在用枚举法解决问题的时候，一定要想清楚这两件事，否则会带来不必要的时间开销。

选择合适的枚举顺序

根据题目判断。比如例题中要求的是最大的符合条件的素数，那自然是从大到小枚举比较合适。

§2.2 模拟算法

§2.2.0 简介

模拟就是用计算机来模拟题目中要求的操作。

模拟题目通常具有码量大、操作多、思路繁复的特点。由于它码量大，经常会出现难以查错的情况，如果在考试中写错是相当浪费时间的。

§2.3 递归算法

§2.3.0 简介

递归（英语：Recursion），在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法。

§2.3.1 要点

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题。求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的。

§2.3.2 缺点

在程序执行中，递归是利用堆栈来实现的。每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧。而栈不是无限大的，当递归层数过多时，就会造成 栈溢出 的后果。

§2.4 分治算法

§2.4.0 简介

分治，字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

§2.4.1 要点

分治算法的核心思想就是「分而治之」。

大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

§2.5 贪心算法

§2.5.0 简介

贪心算法，是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

§2.5.1 要点

最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

§2.6 排序算法

§2.6.0 简介

排序算法（英语：Sorting algorithm）是一种将一组特定的数据按某种顺序进行排列的算法。排序算法多种多样，性质也大多不同。

基于比较的排序算法的时间复杂度下限是$O(n \log n)$ 的。

以下是几种排序算法的比较。
15种排序算法动画演示视频

§2.6.0 种类

选择排序

定义

选择排序（英语：Selection sort）是一种简单直观的排序算法。它的工作原理是每次找出第i小的元素（也就是$A_{i..n}$中最小的元素），然后将这个元素与数组第i个位置上的元素交换。

时间复杂度

$O(n^2)$

代码实现

```cpp
#include <utility>

void selection_sort(int* a, int n) {
  for (int i = 1; i < n; ++i) {
    int ith = i;
    for (int j = i + 1; j <= n; ++j) {
      if (a[j] < a[ith]) {
        ith = j;
      }
    }
    std::swap(a[i], a[ith]);
  }
}
```

```python
def selection_sort(a, n):
    for i in range(1, n):
        ith = i
        for j in range(i + 1, n + 1):
            if a[j] < a[ith]:
                ith = j
        a[i], a[ith] = a[ith], a[i]
```

```java
static void selection_sort(int[] arr, int n) {
    for (int i = 1; i < n; i++) {
        int ith = i;
        for (int j = i + 1; j <= n; j++) {
            if (arr[j] < arr[ith]) {
                ith = j;
            }
        }
        // swap
        int temp = arr[i];
        arr[i] = arr[ith];
        arr[ith] = temp;
    }
}
```

冒泡排序

定义

冒泡排序（英语：Bubble sort）是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢「浮」到数列的顶端，故叫做冒泡排序。

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。

经过i次扫描后，数列的末尾i项必然是最大的i项，因此冒泡排序最多需要扫描$n-1$遍数组就能完成排序。

时间复杂度

$O(n^2)$

代码实现

```
void bubble_sort(int *a, int n) {
  bool flag = true;
  while (flag) {
    flag = false;
    for (int i = 1; i < n; ++i) {
      if (a[i] > a[i + 1]) {
        flag = true;
        int t = a[i];
        a[i] = a[i + 1];
        a[i + 1] = t;
      }
    }
  }
}
```

```
def bubble_sort(a, n):
    flag = True
    while flag:
        flag = False
        for i in range(1, n):
            if a[i] > a[i + 1]:
                flag = True
                a[i], a[i + 1] = a[i + 1], a[i]
```

```
static void bubble_sort(int[] a, int n) {
    boolean flag = true;
    while (flag) {
        flag = false;
        for (int i = 1; i < n; i++) {
            if (a[i] > a[i + 1]) {
                flag = true;
                int t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
}
```

插入排序

定义

插入排序（英语：Insertion sort）是一种简单直观的排序算法。它的工作原理为将待排列元素划分为「已排序」和「未排序」两部分，每次从「未排序的」元素中选择一个插入到「已排序的」元素中的正确位置。

一个与插入排序相同的操作是打扑克牌时，从牌桌上抓一张牌，按牌面大小插到手牌后，再抓下一张牌。

时间复杂度

$O(n^2)$

代码实现

```
void insertion_sort(int arr[], int len) {
  for (int i = 1; i < len; ++i) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}
```

```
def insertion_sort(arr, n):
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

计数排序

定义

计数排序（英语：Counting sort）是一种线性时间的排序算法。

计数排序的工作原理是使用一个额外的数组 C，其中第 i 个元素是待排序数组 A 中值等于 i 的元素的个数，然后根据数组 C 来将 A 中的元素排到正确的位置。1

它的工作过程分为三个步骤：

1. 计算每个数出现了几次；
2. 求出每个数出现次数的 前缀和；
3. 利用出现次数的前缀和，从右至左计算每个数的排名。

> **Tip**
>
> **前缀和**（Prefix Sum）指的是数组中从第一个元素到当前元素之间所有元素的累加和。

时间复杂度

$O(n+w)$，其中w代表待排序数据的值域大小。

代码实现

```
#include <cstring>
constexpr int MAXN = 1010;
constexpr int MAXW = 100010;

int cnt[MAXW], b[MAXN];

int* counting_sort(int* a, int n, int w) {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i]];
  for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
  for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
  return b;
}
```

```
N = W = 100010
n = w = 0
a = b = [0] * N
cnt = [0] * W


def counting_sort():
    for i in range(1, n + 1):
        cnt[a[i]] += 1
    for i in range(1, w + 1):
        cnt[i] += cnt[i - 1]
    for i in range(n, 0, -1):
        b[cnt[a[i]] - 1] = a[i]
        cnt[a[i]] -= 1
```

快速排序

定义

快速排序（英语：Quicksort），又称分区交换排序（英语：partition-exchange sort），简称「快排」，是一种被广泛运用的排序算法。

快速排序的工作原理是通过**分治**的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

时间复杂度

$O(\log n)$

代码实现

```cpp
template <typename T>
int Partition(T A[], int low, int high) {
  int pivot = A[low];
  while (low < high) {
    while (low < high && pivot <= A[high]) --high;
    A[low] = A[high];
    while (low < high && A[low] <= pivot) ++low;
    A[high] = A[low];
  }
  A[low] = pivot;
  return low;
}
template <typename T>
void QuickSort(T A[], int low, int high) {
  if (low < high) {
    int pivot = Partition(A, low, high);
    QuickSort(A, low, pivot - 1);
    QuickSort(A, pivot + 1, high);
  }
}
template <typename T>
void QuickSort(T A[], int len) {
  QuickSort(A, 0, len - 1);
}
```

```python
def quick_sort(alist, first, last):
    if first >= last:
        return
    mid_value = alist[first]
    low = first
    high = last
    while low < high:
        while low < high and alist[high] >= mid_value:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] < mid_value:
            low += 1
        alist[high] = alist[low]
    alist[low] = mid_value
    quick_sort(alist, first, low - 1)
    quick_sort(alist, low + 1, last)
```

归并排序

定义

归并排序（英语：merge sort）是高效的基于比较的稳定排序算法。

归并排序最核心的部分是合并（merge）过程：将两个有序的数组 a[i] 和 b[j] 合并为一个有序数组 c[k]。

从左往右枚举 a[i] 和 b[j]，找出最小的值并放入数组 c[k]；重复上述过程直到 a[i] 和 b[j] 有一个为空时，将另一个数组剩下的元素放入 c[k]。

为保证排序的稳定性，前段首元素小于或等于后段首元素时（a[i] <= b[j]）而非小于时（a[i] < b[j]）就要作为最小值放入 c[k]。

时间复杂度

$O(n \log n)$

代码实现

> 数组实现

```
void merge(const int *a, size_t aLen, const int *b, size_t bLen, int *c) {
  size_t i = 0, j = 0, k = 0;
  while (i < aLen && j < bLen) {
    if (b[j] < a[i]) {  // <!> 先判断 b[j] < a[i]，保证稳定性
      c[k] = b[j];
      ++j;
    } else {
      c[k] = a[i];
      ++i;
    }
    ++k;
  }
  // 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
  for (; i < aLen; ++i, ++k) c[k] = a[i];
  for (; j < bLen; ++j, ++k) c[k] = b[j];
}
```

> 指针实现

```
void merge(const int *aBegin, const int *aEnd, const int *bBegin,
           const int *bEnd, int *c) {
  while (aBegin != aEnd && bBegin != bEnd) {
    if (*bBegin < *aBegin) {
      *c = *bBegin;
      ++bBegin;
    } else {
      *c = *aBegin;
      ++aBegin;
    }
    ++c;
  }
  for (; aBegin != aEnd; ++aBegin, ++c) *c = *aBegin;
```

```
  for (; bBegin != bEnd; ++bBegin, ++c) *c = *bBegin;
}
```

```python
def merge(a, b):
    i, j = 0, 0
    c = []
    while i < len(a) and j < len(b):
        # <!> 先判断 b[j] < a[i]，保证稳定性
        if b[j] < a[i]:
            c.append(b[j])
            j += 1
        else:
            c.append(a[i])
            i += 1
    # 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
    c.extend(a[i:])
    c.extend(b[j:])
    return c
```

堆排序

定义

堆排序（英语：Heapsort）是指利用 二叉堆 这种数据结构所设计的一种排序算法。堆排序的适用数据结构为数组。

堆排序的本质是建立在堆上的选择排序。首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；

之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；

以此类推，在第 n-1 次操作后，整个数组就完成了排序。

时间复杂度

$O(n \log n)$

代码实现

```
void sift_down(int arr[], int start, int end) {
  // 计算父结点和子结点的下标
  int parent = start;
  int child = parent * 2 + 1;
  while (child <= end) {  // 子结点下标在范围内才做比较
    // 先比较两个子结点大小，选择最大的
    if (child + 1 <= end && arr[child] < arr[child + 1]) child++;
    // 如果父结点比子结点大，代表调整完毕，直接跳出函数
    if (arr[parent] >= arr[child])
      return;
    else {  // 否则交换父子内容，子结点再和孙结点比较
      swap(arr[parent], arr[child]);
      parent = child;
      child = parent * 2 + 1;
    }
  }
```

```
}

void heap_sort(int arr[], int len) {
  // 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
  for (int i = (len - 1 - 1) / 2; i >= 0; i--) sift_down(arr, i, len - 1);
  // 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素），直到排序完毕
  for (int i = len - 1; i > 0; i--) {
    swap(arr[0], arr[i]);
    sift_down(arr, 0, i - 1);
  }
}
```

```
def sift_down(arr, start, end):
    # 计算父结点和子结点的下标
    parent = int(start)
    child = int(parent * 2 + 1)
    while child <= end:  # 子结点下标在范围内才做比较
        # 先比较两个子结点大小，选择最大的
        if child + 1 <= end and arr[child] < arr[child + 1]:
            child += 1
        # 如果父结点比子结点大，代表调整完毕，直接跳出函数
        if arr[parent] >= arr[child]:
            return
        else:  # 否则交换父子内容，子结点再和孙结点比较
            arr[parent], arr[child] = arr[child], arr[parent]
            parent = child
            child = int(parent * 2 + 1)


def heap_sort(arr, len):
    # 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
    i = (len - 1 - 1) / 2
    while i >= 0:
        sift_down(arr, i, len - 1)
        i -= 1
    # 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素），直到排序完毕
    i = len - 1
    while i > 0:
        arr[0], arr[i] = arr[i], arr[0]
        sift_down(arr, 0, i - 1)
        i -= 1
```

§2.7 查找算法

§2.7.0 简介

> **Important**
>
> 本页面将简要介绍二分查找

二分查找（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是用来在一个有序数组中查找某一元素的算法。

§2.7.1 过程

以在一个升序数组中查找一个数为例。

它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找。

§2.7.1 时间复杂度

$O(\log n)$

§2.7.2 代码实现

> **Tip**
>
> C++ 标准库中实现了查找首个不小于给定值的元素的函数 std::lower_bound 和查找首个大于给定值的元素的函数 std::upper_bound，二者均定义于头文件 <algorithm> 中。
>
> 二者均采用二分实现，所以调用前必须保证元素有序。

```
int binary_search(int start, int end, int key) {
  int ret = -1;  // 未搜索到数据返回-1下标
  int mid;
  while (start <= end) {
    mid = start + ((end - start) >> 1);  // 直接平均可能会溢出，所以用这个算法
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else {  // 最后检测相等是因为多数搜索情况不是大于就是小于
      ret = mid;
      break;
    }
  }
  return ret;  // 单一出口
}
```

§2.8 数据结构与算法

> 数据结构是程序的骨架，而算法则是程序的灵魂。

§2.8.0 数据结构的定义

数据结构（英语：data structure）：带有结构特性的数据元素的集合。

简单来说，**算法**就是解决问题的方法或者过程。如果我们把问题看成是函数，那么算法就是将输入转换为输出的过程。**数据结构**是数据的计算机表示和相应的一组操作。**程序**则是算法和数据结构的具体实现。

展开来讲，**数据结构研究的是数据的逻辑结构、物理结构以及它们之间的相互关系，并对这种结构定义相应的运算，设计出相应的算法，并确保经过这些运算以后所得到的新结构仍保持原来的结构类型。

对于数据结构，我们可以按照数据的**逻辑结构**和**物理结构**来进行分类。

§2.8.0 逻辑结构

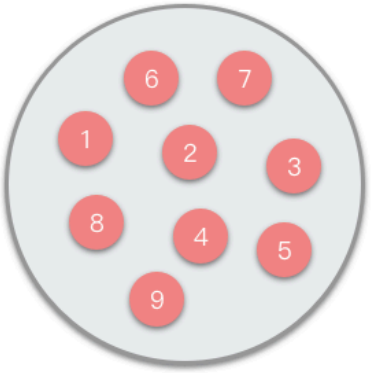
定义

逻辑结构（英语：logical structure）：数据元素之间的相互关系。

集合结构

集合结构：数据元素同属于一个集合，除此之外无其他关系。

集合结构中的数据元素是无序的，并且每个数据元素都是唯一的，集合中没有相同的数据元素。集合结构很像数学意义上的「集合」。



python中的集合（set）是一种集合结构数据类型

```
MusicGames = {"Arcaea", "Phigros", "maimaiDX", "CHUNITHM"}
MusicGames.add("Rizline")
MusicGames.remove("Phigros")
if "Arcaea" in MusicGames:
    print("Remember to play Arcaea")
```

线性结构

线性结构：数据元素之间是**一对一**关系。

线性结构中的数据元素（除了第一个和最后一个元素），左侧和右侧分别只有一个数据与其相邻。线性结构类型包括：数组、链表，以及由它们衍生出来的栈、队列、哈希表。



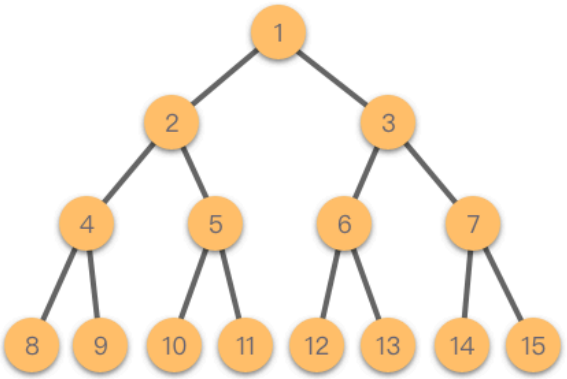
python中的列表（list）是一种集合结构数据类型

```
MusicGames = ["Arcaea", "Phigros", "maimaiDX", "CHUNITHM"]
MusicGames.append("Rizline")
del MusicGames[1]
print("Remember to play " + MusicGames[0])
```

树形结构

树形结构：数据元素之间是**一对多**关系。

最简单的树形结构是二叉树。这种结构可以简单的表示为：根， 左子树， 右子树。 左子树和右子树又有自己的子树。当然除了二叉树，树形结构类型还包括：多叉树、字典树等。

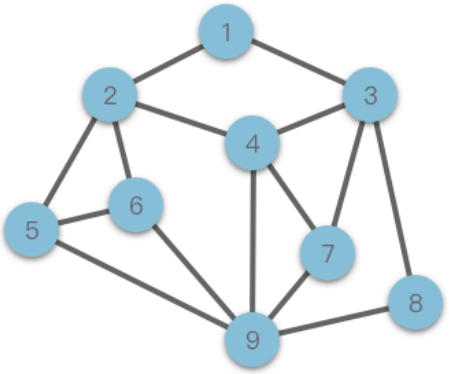


图形结构

图形结构：数据元素之间是**多对多**关系。

图形结构是一种比树形结构更复杂的非线性结构，用于表示物件与物件之间的关系。一张图由一些小圆点（称为**顶点**或**结点**）和连结这些圆点的直线或曲线（称为**边**）组成。

在图形结构中，任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。图形结构类型包括：无向图、有向图、连通图等



§2.8.1 物理结构

定义

物理结构（**physical structure**）：数据的逻辑结构在计算机中的存储方式。

顺序存储结构

顺序存储结构（sequential storage structure）：将数据元素存放在一片地址连续的存储单元里，数据元素之间的逻辑关系通过数据元素的存储地址来直接反映。



在顺序存储结构中，逻辑上相邻的数据元素在物理地址上也必然相邻 。

这种结构的优点是：

- 简单、易理解，且实际占用最少的存储空间。

缺点是：

- 需要占用一片地址连续的存储单元；
- 并且存储分配要事先进行；
- 另外对于一些操作的时间效率较低（移动、删除元素等操作）。

链式存储结构

链式存储结构（linked storage structure）：将数据元素存放在任意的存储单元里，存储单元可以连续，也可以不连续。



链式存储结构中，逻辑上相邻的数据元素在物理地址上可能相邻，可也能不相邻。其在物理地址上的表现是随机的。

链式存储结构中，一般将每个数据元素占用的若干单元的组合称为一个链结点。每个链结点不仅要存放一个数据元素的数据信息，还要存放一个指出这个数据元素在逻辑关系的直接后继元素所在链结点的地址，该地址被称为指针。换句话说，数据元素之间的逻辑关系是通过指针来间接反映的。

这种结构的优点是：

- 存储空间不必事先分配，在需要存储空间的时候可以临时申请，不会造成空间的浪费；
- 一些操作的时间效率远比顺序存储结构高（插入、移动、删除元素）。

缺点是：

- 不仅数据元素本身的数据信息要占用存储空间，指针也需要占用存储空间，链式存储结构比顺序存储结构的空间开销大。

§3 常见开发问题及解决方法

§3.0 Python常见报错及解决方法

§3.0.0 SyntaxError（语法错误）

常见表现：

- `SyntaxError: invalid syntax`
- `SyntaxError: EOL while scanning string literal`

常见原因：

- 忘记闭合括号、引号
- 使用了中文标点符号
- 缩进错误
- 错误的关键字使用

解决方法：

1. 检查报错行及其上一行的语法
2. 使用代码编辑器的高亮功能帮助识别
3. 示例修复：

```
# 错误示例
print('Hello world"

# 正确示例
print('Hello world')
```

§3.0.1 NameError（名称错误）

常见表现：

- `NameError: name 'xxx' is not defined`

常见原因：

- 变量/函数名拼写错误
- 变量在使用前未定义
- 未导入需要的模块

解决方法：

1. 检查变量名是否正确拼写
2. 确保变量在使用前已定义
3. 示例修复：

```
# 错误示例
print(messege)  # 'message'拼写错误

# 正确示例
message = "Hello"
print(message)
```

§3.0.2 TypeError（类型错误）

常见表现：

- `TypeError: can only concatenate str (not "int") to str`
- `TypeError: 'xxx' object is not callable`

常见原因：

- 不同类型数据间的非法操作
- 误将变量名用作函数名

解决方法：

1. 检查变量类型(type(var)查看)
2. 进行必要的类型转换
3. 示例修复：

```
# 错误示例
age = 25
print("Age: " + age)  # 不能直接拼接字符串和整数

# 正确示例
age = 25
print("Age: " + str(age))
```

§3.0.3 IndexError（索引错误）

常见表现：

- `IndexError: list index out of range`

常见原因：

- 访问列表/元组不存在的索引
- 空列表访问

解决方法：

1. 检查列表长度(len(list))
2. 添加边界检查
3. 示例修复：

```python
# 错误示例
my_list = [1, 2, 3]
print(my_list[3])  # 有效索引是0-2

# 正确示例
my_list = [1, 2, 3]
if len(my_list) > 3:
    print(my_list[3])
```

§3.0.4 KeyError（键错误）

常见表现：

- `KeyError: 'xxx'`

常见原因：

- 访问字典不存在的键

解决方法：

1. 使用dict.keys()检查键是否存在
2. 使用dict.get()方法提供默认值
3. 示例修复：

```python
# 错误示例
my_dict = {'name': 'Alice'}
print(my_dict['age'])  # 'age'键不存在

# 正确示例
my_dict = {'name': 'Alice'}
print(my_dict.get('age', 'N/A'))  # 返回'N/A'如果键不存在
```

§3.1 Debug工具简介及使用

§3.1.0 Pyhon Debugger（PDB）

PDB是Python自带的调试工具，可以帮助开发者：逐行执行代码、查看变量状态、设置断点、分析程序流程（及实现一些黑魔法）。

通常来说，我们通过命令行来启动PDB，即：

```
python -m pdb your_script.py
```

而仅仅使用PDB运行Python代码并不能起到调试作用。为进行调试，你还需要给程序添加断点（Breakpoint），如下所示：

```python
import pdb; pdb.set_trace()  # Python 3.7以前
# 或
breakpoint()  # Python 3.7+
```

> **Tip**
>
> 若程序中有`breakpoint`命令，则运行到此行命令时，即使没有在启动时添加`-m pdb`参数，也会自动调起PDB。

PDB的具体使用过程此处略去：一方面，现代IDE的Debug功能相较命令行更加直观；另一方面，其具体使用仅通过文本难以完全概述。如有需要，可以参见：《【python】来学学debugger吧，不能只会用print调试呀！》| 哔哩哔哩：https://www.bilibili.com/video/BV1La4y1T7Y5

虽然此处不对其进行讲解，但仍给出常用PDB命令如下：

命令	说明
l(list)	显示当前代码段
n(next)	执行下一行
s(step)	进入函数内部
c(continue)	继续执行直到下一个断点
p(print)	打印变量值
q(quit)	退出调试
h(help)	查看帮助

§3.1.1 PyCharm Debugger使用

基本步骤

1. 在代码行号旁点击设置断点（红色圆点）
2. 点击右上角的"Debug"按钮（绿色虫子图标）
3. 使用调试工具栏控制执行：
    - Step Over (F8)：单步执行
    - Step Into (F7)：进入函数
    - Step Out (Shift+F8)：跳出函数
    - Resume Program (F9)：继续到下一个断点

实用功能

- 查看变量值：在Debug窗口的"Variables"面板
- 计算表达式：在Debug窗口的"Watches"面板
- 条件断点：右键点击断点设置条件

快捷键

操作	快捷键	说明
继续	F9	执行到下一个断点
单步跳过	F8	执行当前行，不进入函数
单步进入	F7	进入当前行的函数内部
单步跳出	Shift+F8	跳出当前函数
运行到光标	Alt+F9	执行到光标位置
重新运行	Ctrl+F5	重新开始调试
停止	Ctrl+F2	结束调试
快速计算	Alt+F8	快速计算任意表达式

高级功能

- 值渲染：自定义对象显示方式
- 帧跳转：在调用栈中自由切换上下文
- 多线程调试：支持线程切换和单独控制
- 远程调试：支持SSH、Docker等远程解释器

§3.1.2 VSCode Python Debugger

基本配置

1. 确保已安装Python扩展（Microsoft官方Python扩展）
2. 打开要调试的Python文件
3. 点击左侧活动栏的"运行和调试"图标（或按`Ctrl+Shift+D`）

设置断点

- 在代码行号左侧点击即可设置/取消断点（红色圆点）
- 右键点击断点可设置条件断点或日志点

启动调试

1. 点击顶部调试工具栏的绿色箭头（或按`F5`）
2. 首次调试需要选择调试配置：
    - 通常选择"Python文件"调试当前打开的文件
    - 其他选项包括Django、Flask等框架调试

快捷键

操作	快捷键	说明
继续	F5	执行到下一个断点
单步跳过	F10	执行当前行，不进入函数
单步进入	F11	进入当前行的函数内部
单步跳出	Shift+F11	跳出当前函数
重启	Ctrl+Shift+F5	重新开始调试
停止	Shift+F5	结束调试

高级功能

- 在`.vscode/launch.json`中自定义调试配置
- 使用"调试控制台"交互式执行代码
- 支持远程调试和Docker容器内调试

§3.2 其他常见Bug与排查方法

§3.2.0 逻辑错误

通常来说，逻辑错误的导致原因有两个：1. 忽视了某些语言特色，导致了逻辑错误；2. 算法有误。

表现特点

程序能运行但结果不正确。

排查方法

1. 添加print语句输出关键变量
2. 编写单元测试验证小功能块
3. 使用断言(assert)检查假设
4. 使用Debugger进行调试

§3.2.1 无限循环

无限循环也可以当作是一种逻辑错误。顾名思义，此问题即为由于逻辑问题等导致的无限而不终止的循环，最后导致程序卡死。

表现特点

程序卡死，CPU占用高。

排查/验证方法

1. 在循环中适当添加print语句
2. 测试多种输入数据/交互逻辑，从而找到发生无限循环的输入，从而推断导致无限循环可能的原因

解决方法

1. 检查循环条件是否能被满足
2. 添加临时计数器强制退出（不推荐）

```
max_iter = 1000
count = 0
while condition and count < max_iter:
    # 循环体
    count += 1
```

§3.2.2 模块导入错误

通常来说，模块导入错误直接从报错中即可得知。

常见表现

- `ModuleNotFoundError: No module named 'xxx'`
- `ImportError: cannot import name 'xxx'`

解决方法

1. 利用`pip list`确保模块已安装
2. 检查导入的名称是否正确（点名批评某些`pip`包名和`module`包名不同的包）
3. 检查运行环境，尤其是在`venv`虚拟环境中的时候
4. 检查文件名是否与标准库冲突（尤其是自己写的库时）

§3.2.3 文件路径问题

常见表现

- `FileNotFoundError: [Errno 2] No such file or directory`

解决方法

1. 使用绝对路径
2. 使用os.path检查路径是否存在

```
import os
if os.path.exists(file_path):
    # 处理文件
```

3. 使用try-except处理异常

```
try:
    with open('file.txt') as f:
        content = f.read()
except FileNotFoundError:
    print("文件不存在")
```

4. 检查Python运行路径

§3.2.4 内存问题

常见表现

程序运行越来越慢，最终崩溃

解决方法

1. 检查是否有无限增长的数据结构
2. 使用生成器代替大列表，避免类似`list(range(...))`的表达
3. 及时关闭文件、数据库连接
4. 使用内存分析工具如`memory_profiler`

§3.3 通用调试技巧

此处给出一些一般性的调试技巧，此类技巧与编程语言类型无关且实用。

1. 缩小范围：通过注释代码定位、猜测可能出现问题区域
2. 多种条件测试：通过多种不同输入的结果，猜测可能的问题诱因
3. 二分查找法：利用二分搜索，逐步缩小问题范围
4. 橡皮鸭调试法：向他人（或橡皮鸭）解释代码的逻辑和运行过程，尝试在此过程中发现未被注意到的错误
5. 版本对比：与之前正常工作的版本比较（建议在必要时，复刻所有运行环境，包括其他使用的库、运行目录下的文件甚至是运行系统版本等）
6. 查阅文档：官方文档常包含常见问题解答（FAQ），也可以从官方文档中观察是否存在一些隐性的奇怪运行逻辑，导致Bug

§4 后言

后言没什么好写的。

期待以后的社员能给这篇讲义中很多略讲甚至未讲的部分进行补充，并且在需要的地方添加直观的配图。