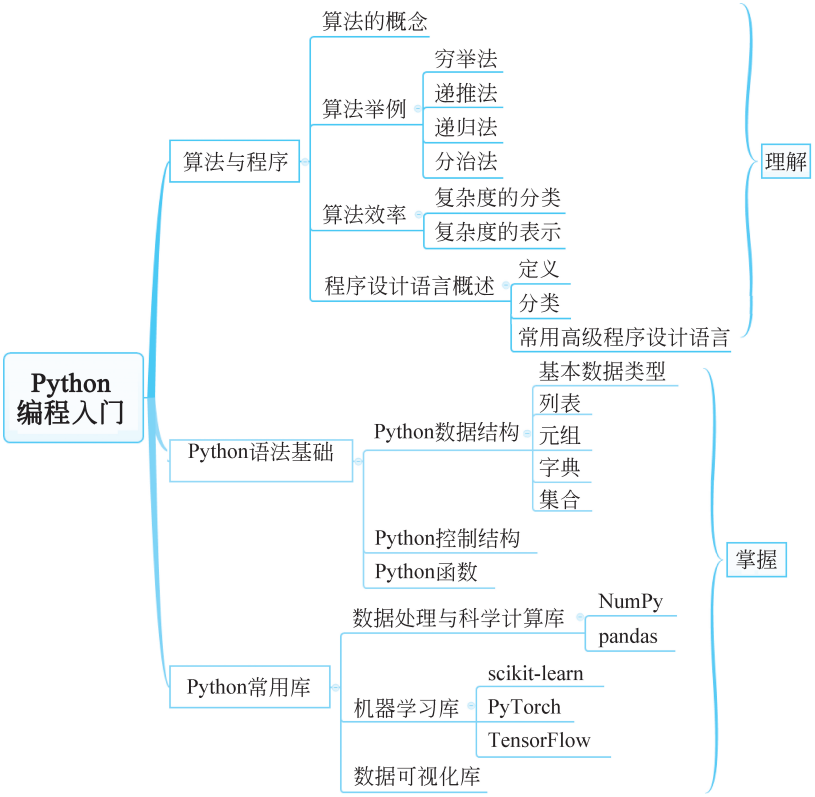
第3章

Python 编程入门



本章首先阐述计算机算法与程序的基本概念，继而介绍面向人工智能领域的Python 语法基础。

3.1 算法与程序

本节介绍算法的概念、效率与程序设计语言的基本定义和分类。

3.1.1 算法的概念

算法(Algorithm)是对问题解决方案的形式化描述，由一系列为解决特定问题而设计的明确、有限且可执行的步骤或规则构成。其本质是通过系统化方法刻画问题求解的策略机制，即在给定初始状态或输入数据的前提下，通过一系列步

骤得到确定的解决方案。在中国古代数学文献中，算法的思想早有体现，例如西汉时期的《周髀算经》所记载的勾股定理计算方法，已蕴含了明确的算法化思想。

算法需满足有穷性（在有限步内终止）、确定性（每一步结果唯一）、可行性（所有操作均能有效执行，并可通过基本运算有限次完成），可以有零个或多个输入，必须至少有一个输出。此外，优秀算法通常还具备鲁棒性，能妥善处理异常情况。

1. 控制结构

算法的控制结构用以定义其执行流程。常见的算法控制结构主要包括三种类型：顺序结构、选择结构（亦称为条件结构）以及循环结构。

- 顺序结构。在此结构中，算法步骤严格依照预定的先后顺序依次执行，构成了最为基础的程序执行流程。
- 选择结构。该结构依据条件表达式的布尔值，选择执行不同的代码分支，从而实现程序逻辑的条件化转向。
- 循环结构。此结构在循环条件满足时重复执行特定代码段，适合实现重复性计算的任务，直至条件不再满足时终止循环。

2. 算法要素

算法通常由输入（Input）、处理（Process）和输出（Output）三个核心部分构成。输入是算法执行的起点，为处理提供所需的数据对象；处理则是从输入到输出的有序指令集合，其设计须保证正确性与效率，通过清晰、可操作的任务序列确保算法能够被准确实现与执行；输出是算法最终产生的结果，体现了处理的目标与效果。

算法的基本运算是指算法中的基本操作或计算步骤，主要包括以下类别。

- 算术运算。对数值执行数学计算，如加、减、乘、除等。
- 关系运算。比较两个值的大小或相等性，并返回布尔值（真或假）。
- 逻辑运算。对布尔值进行与（AND）、或（OR）、非（NOT）等操作，常用于条件判断。
- 赋值运算。将表达式的值存储至变量中。
- 数据访问与移动。包括数据的输入、输出、读取或修改存储单元中的内容。

3.1.2 算法举例

在计算机科学领域，算法不仅是解决问题的具体步骤，更体现了一种系统化的思维范式。常见的算法设计方法包括穷举法、递推法、递归法、分治法等。这些方法体现了从“暴力尝试”到“智能构造”的策略层次，构成了从基础计算到人工智能决策的核心逻辑基础。下文将结合典型示例，简要阐述各类方法的基本思想及其应用场景。

1. 判断素数（穷举法）

判定一个大于 1 的自然数是否为素数（亦称质数），最直接的方法是穷举法（Brute Force）。其基本思路在于：依次使用 $2 \sim n-1$（或该数平方根）的所有整数去除目标数 n。若存在任一整数能够整除 n，则 n 为合数；若所有整数均不能整除 n，则 n 为素数。后者基于数论中的一个基本结论：若一个数存在大于其平方根的因子，则必然存在对应的小于其平方根的因子。尽管穷举法在输入规模较大时效率较低，但其逻辑简洁、正确性明确，是理解“枚举所有可能解”这一基础算法策略的典型范例。这两种算法的伪代码实现如表 3.1 所示。

表 3.1　判断素数算法的伪代码

算法 1：2～$n-1$	算法 2：2～$\sqrt{n}$

i ← 2
while i < n do
　　if n mod i == 0 then
　　　　return False
　　end if
　　i ← i + 1
end while
return True
	i ← 2
	while i * i <= n do
	　　if n mod i == 0 then
	　　　　return False
	　　end if
	　　i ← i + 1
	end while
	return True

2. 辗转相除法求最大公因子(递推法)

辗转相除法又称欧几里得算法，是一种基于递推关系求解两个非负整数最大公因子的高效方法。其核心思想是：两个整数的最大公因子等于其中较小数与两数相除余数的最大公因子。具体步骤为：用较大数除以较小数，将所得余数替换为原除数，原除数替换为被除数，重复此过程直至余数为零，此时除数即为两数的最大公因子。该算法通过递推公式 gcd(a,b)＝gcd(b,a mod b)将问题规模逐次缩减，最终在有限步内得到确定解，过程如表 3.2 所示。

表 3.2　辗转相除法的示例

数 a	数 b	余　　数
a	b	r
48	18	12
18	12	6
12	**6**	0

3. 斐波那契数列(递归法)

斐波那契数列是以递归方式定义的经典整数序列，其递推关系为 $F(n)=F(n-1)+F(n-2)$，初始条件 $F(0)=0$，$F(1)=1$。递归求解斐波那契数列时，将原问题分解为两个规模更小的同类子问题，通过递归调用逐层分解直至基准情形。例如，计算 $F(5)$需先计算 $F(4)$与 $F(3)$，而 $F(4)$又依赖 $F(3)$和 $F(2)$，以此类推，计算过程如表 3.3 所示。尽管递归表达直观地体现了数学定义，但存在重复计算问题，可通过记忆化或动态规划优化效率。

表 3.3　斐波那契数列的递推计算过程

$F(n-2)$	$F(n-1)$	$F(n)$	n
1	1	2	3
1	2	3	4
2	3	5	5
3	5	8	6
5	8	13	7

4. 快速排序(分治法)

快速排序(Quick Sort)是一种高效的排序算法,基于分治法(Divide and Conquer)的思想。它的核心是通过选择一个基准元素(pivot),将列表分为两部分:一部分小于基准元素,另一部分大于基准元素,递归排序这两部分。表 3.4 给出了快速排序的伪代码和具体的数据序列快速排序过程。

表 3.4 快速排序的伪代码和具体的数据序列快速排序过程

伪 代 码	举 例

QuickSort(A, low, high)
  if low < high then
  mid← Partition(A, low, high)
  QuickSort(A, low, mid − 1)
  QuickSort(A, mid + 1, high)
  end if
Partition(A, low, high)
  pivot ← A[low]
  i ← low
  j ←high
  while i < j
  while i < j && A[j] >= pivot j−−
  while i < j && A[i] <= pivot i++
  A[i] ↔ A[j]
  A[low] =A[i]
  A[i] = pivot
	22, 64, 12, 90, 37, 92, 53, 68
	[12, 22, 64, 90, 37, 92, 53, 68]
	(处理整个数组)
	
	[12, 22, 64, 90, 37, 92, 53, 68]
	(处理[12])
	[12, 22, 53, 37, 64, 92, 90, 68]
	(处理[53, 37, 64, 92, 90, 68])
	[12, 22, 37, 53, 64, 92, 90, 68]
	(处理[37])
	[12, 22, 37, 53, 64, 68, 90, 92]
	(处理[64, 68, 90, 92])
	[12, 22, 37, 53, 64, 68, 90, 92]
	(处理[68, 90, 92])
	[12, 22, 37, 53, 64, 68, 90, 92]
	(处理[90, 92])

5. 二分查找(分治法)

折半查找,也称二分查找,用于对有序序列进行快速查找,通过比较中间元素与目标值,将搜索范围逐次减半直至找到目标或确认不存在。

假设数据为升序(由小到大),进行折半查找的思路是:先确定整个查找区间的中间位置 mid = (left + right)/2;用待查关键字值与中间位置的关键字值进行比较;若相等,则查找成功;若大于中间值,则在后(右)半个区域继续进行折半查找;若小于中间值,则在前(左)半个区域继续进行折半查找。重复上述步骤,对缩小区域再按折半公式,要么查找成功,要么查找失败。

【例 3.1】 使用折半查找算法对序列[12, 22, 37, 53, 64, 68, 90, 92]进行查找,关键字为 17,查找过程如表 3.5 所示。

表 3.5 折半查找的过程

序 号	1	2	3	4	5	6	7	8	查找元素 17
数据列表	12	22	37	53	64	68	90	92	
第 1 次	L			**mid**				R	mid = 4
	12	22	37	53	64	68	90	92	
第 2 次	L	**mid**							mid = 2
	12	22	37	53	64	68	90	92	

续表

序　　号	1	2	3	4	5	6	7	8	查找元素 17
第 3 次	**mid/l**								mid= 2
	12	22	37	53	64	68	90	92	
									low = 2，high =1，查找失败

3.1.3　算法效率

算法效率表征算法在实际运行过程中对计算资源的利用效能，而算法复杂度则为其提供了理论上的上界约束与指导，使得我们能够预测算法在大规模输入下的性能趋势。

1. 复杂度的分类

算法复杂度是指算法被实现为可执行程序后，其运行时所消耗的资源与问题规模之间的函数关系，主要包括时间复杂度（即算法执行所需的时间开销）和空间复杂度（即算法运行过程中所占用的内存空间大小）。

- **时间复杂度**。指算法运行时间随输入数据规模变化的关系，通常记为 $T(n)$，其中 n 表示问题的规模，$T(n)$代表基本操作的执行次数。
- **空间复杂度**。指算法在运行过程中临时占用的存储空间大小随输入规模变化的量度，通常记为 $S(n)$，n 为问题的规模。

2. 复杂度的表示

算法的时间复杂度 $T(n)$与空间复杂度 $S(n)$均以输入规模 n 的函数形式进行度量，普遍采用大 O 记法表示为 $O(f(n))$。其中，n 为数据规模，$f(n)$为关于 n 的某个函数，常见形式如 $f(n)=2n+1$、$f(n)=n^2$、$f(n)=\log n$ 等。

算法复杂度 $O(f(n))$与 n 的对应关系可视为一条函数曲线。随着 n 逐渐增大至足够大时，忽略对曲线趋势影响较小的因素（如常数项、系数等），保留决定算法复杂度量级的如图 3.1 所示的核心项，例如多项式最高次项、对数项等。推导大 O 阶的一般方法如下。

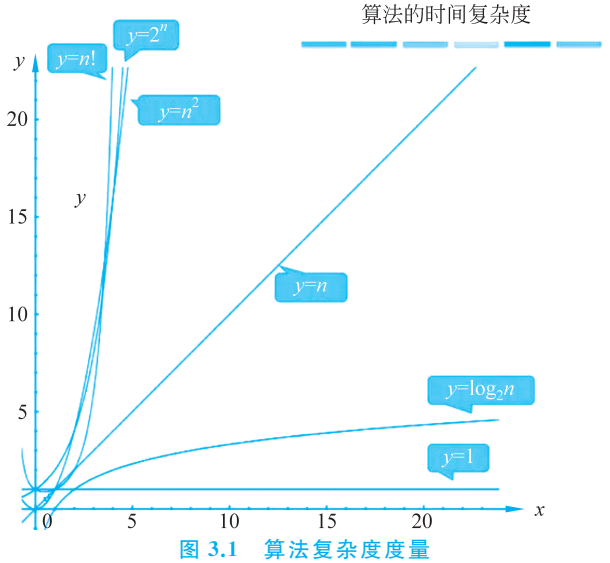


图 3.1　算法复杂度度量

图 3.1 彩图

（1）用常数 1 取代运行时间中的所有加法常数。

（2）在修改后的运行次数函数中，只保留最高阶项。

（3）如果最高阶项存在且系数不是 1，则去除与这个项相乘的常数。得到的结果即为大 O 阶。

【例 3.2】 分析表 3.1 中求解素数的两种算法，计算其时间复杂度，并通过示例进行对比。

算法 1 的时间复杂度：$O(n)$。在最坏的情况下（当 n 是质数时），内层循环需要执行 $(n-2)$ 次。循环的执行次数与输入规模 n 成线性正比关系。$O(n)$称为线性时间复杂度。

算法 2 的时间复杂度：$O(\sqrt{n})$。循环的终止条件是 $i*i<=n$，即 $i<=\sqrt{n}$。因此，在最坏的情况下，循环最多执行$\sqrt{n}$次。循环次数与 n 的平方根相关。$O(\sqrt{n})$称为平方根时间复杂度。

两种算法的时间复杂度对比结果如表 3.6 所示，可视化对比如图 3.2 所示，$O(\sqrt{n})$的时间复杂度的增长速度远慢于 $O(n)$。随着问题规模 n 的增大，$O(\sqrt{n})$算法所需的计算步骤数较 $O(n)$算法显著减少。

表 3.6 求素数的两种算法的时间复杂度对比

输入规模 n	$O(n)$操作次数	$O(\sqrt{n})$操作次数	速 度 倍 差
10	10	≈ 3.2	~3×
100	100	10	10×
1000	1000	≈ 31.6	~32×
10000	10000	100	100×
100000	100000	≈ 316	~316×

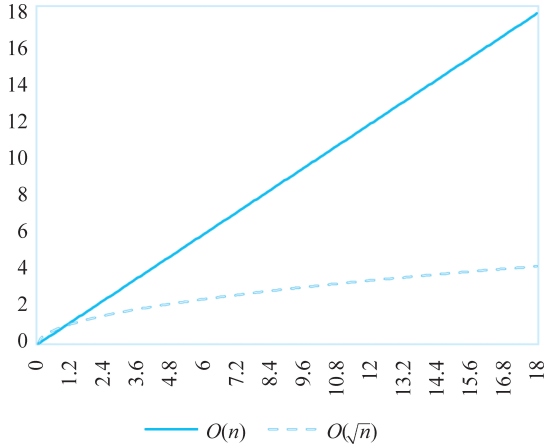


图 3.2 两种算法执行的时间复杂度可视化的对比

为什么算法的时间复杂度不用时间单位衡量？

算法的时间复杂度之所以不用秒、毫秒等时间单位衡量，是因为它本质上关注的是算法执行时间随输入规模增长的趋势，而非一次具体的运行耗时。使用时间单位会使其受

到计算机硬件性能、编程语言效率、操作系统调度、编译器优化，乃至程序当前运行环境（如后台负载）等诸多无关变量的干扰，导致结果完全失去普适性和可比性。因此，时间复杂度采用大 O 表示法进行抽象，只描述计算步骤相对于输入数据量的增长级数，从而剥离了各种外部因素，纯粹地揭示了算法本身效率的内在规律。

3.1.4　程序设计语言概述

程序设计语言作为算法实现的媒介，将抽象的计算逻辑转换为计算机可执行的具体指令序列。

1. 程序设计语言的定义

程序设计语言是一种用于人机交互的标准化工具，旨在将人类思维转换为计算机可执行的指令序列。该语言由语法规则、语义定义与语用规范共同构成，通过特定符号系统（如关键字、运算符和数据结构）表达算法逻辑。

概括而言，程序设计语言是计算机程序的书写语言，其核心价值在于降低人机交互的复杂度：开发者无须掌握硬件细节，即可采用近似自然语言的表达方式描述问题的求解过程。在人工智能领域，这一特性使编程语言成为构建算法模型与训练机器学习系统的基础载体。例如，Python 凭借其语法简洁性与表达力，已成为人工智能开发的通用工作语言。

2. 程序设计语言的分类

程序设计语言依据抽象程度与功能特性，可分为三类：机器语言、汇编语言和高级语言。

机器语言采用二进制代码（0 和 1）直接表征指令。该语言直接面向计算机硬件，虽可被计算机直接执行且运行效率高，但指令缺乏直观性、难以理解，编程难度较大、易出错，且所编程序不具备可移植性，仅适用于特定型号的机器。机器语言对编程人员要求较高，需熟知计算机的体系结构，因此目前极少直接使用机器语言进行编程。

汇编语言借助助记符号（如 ADD、MOV）编写程序，无法被计算机直接执行，须通过“汇编程序”汇编为机器指令后方可识别与执行。汇编语言属于低级语言，直接操作计算机硬件。尽管其使用不便、流程烦琐，且要求编程人员熟悉计算机的内部结构，但程序占用内存空间少、执行速度快，适用于系统软件与过程控制软件的开发。

高级语言采用接近自然语言的语法（如 if 条件语句、for 循环结构），直观易懂、易于掌握，且无须编程人员了解计算机结构。高级语言编写的程序独立于计算机硬件，具备良好的可移植性，可在多种类型的计算机上运行。代表语言包括 Python、Java、C++ 等。在人工智能领域，高级语言是开发的主流工具：Python 凭借其简洁语法、丰富的 AI 库（如 NumPy、TensorFlow）及跨平台特性，成为算法实现与数据处理的首选，显著降低了人工智能技术的应用门槛。

表 3.7 给出了 $x=x+1$ 的三种程序设计语言的代码对比，可见高级程序设计语言 Python 的代码最简洁，机器语言难以理解。

表 3.7　三种程序设计语言的代码对比（$x=x+1$）

机器语言	10000011 （0x83） （add）	11000011 （0xC3） （EAX 寄存器）	00000001 （0x01） （01）
汇编语言	add eax，1		
Python	$x=x+1$		

3. 常用的高级程序设计语言

高级程序设计语言与机器语言和汇编语言相比，具有抽象层次高、可读性强、可移植性好、开发效率高等特点。程序开发人员无须直接操作寄存器或内存地址，而是通过变量、函数、类、模块等抽象结构来组织程序逻辑。自 20 世纪中期以来，高级语言不断发展，从早期的科学计算语言到面向对象语言，再到现代脚本语言与系统语言，形成了多样化生态。

- Python：语法简洁、库生态丰富，广泛应用于数据分析、人工智能、自动化脚本和后端开发。
- C：接近硬件、执行效率高，主要用于操作系统、嵌入式系统和底层系统软件开发。
- C++：在 C 的基础上支持面向对象和泛型编程，常用于高性能软件、游戏引擎和大型工程系统。
- Java：强调跨平台与面向对象的特性，广泛用于企业级应用、Android 开发和大型分布式系统。
- C#：与.NET 平台紧密结合，主要应用于 Windows 应用开发、企业系统及 Unity 游戏开发。
- JavaScript：浏览器原生支持的脚本语言，是 Web 前端开发和现代 Web 应用交互的核心技术。
- Visual Basic：语法简单、易于上手，常用于快速开发 Windows 桌面应用和办公自动化程序。
- SQL：专门用于数据库查询与管理，是关系数据库系统中进行数据操作和分析的标准语言。
- R：专注于统计计算和数据可视化，广泛应用于数据分析、学术研究和统计建模领域。
- Perl：擅长文本处理与脚本编写，常用于系统管理、网络编程和早期的 Web 开发。

图 3.3 展示的是近年来排名前十的高级程序设计语言 TIOBE 社区指数，从图中可以看出，2025 年 7 月 Python 的活跃占比达到 26.98%，创历史新高，在众多主流语言中保持领先地位。其持续上升的原因在于：语法简洁、开发效率高、标准库完备、第三方生态成熟，能够覆盖 Web 开发、自动化运维、数据分析、科学计算等多类应用场景。尤其是在人工智能时代，Python 凭借完善的数值计算与机器学习生态（如深度学习框架和数据处理工具），成为人工智能与大数据开发的核心语言。在众多实际开发、算法实现、实验验证以及系统部署等环节，Python 都提供了成熟而高效的解决方案，已成为人工智能技术工程化落地的重要支撑平台。

图 3.3　近年来排名前十的高级程序设计语言 TIOBE 社区指数

图 3.3 彩图

3.2　Python 语法基础

Python 语言以其清晰的语法、简洁的结构而著称，具备卓越的可读性与跨平台兼容性。凭借其丰富的标准库与庞大的第三方生态系统，Python 能够高效地支持数据处理、模型构建及系统开发等一系列任务。在人工智能领域，诸如 TensorFlow、PyTorch 等主流深度学习框架均将 Python 作为核心接口语言，从而构建了完善的工具链与活跃的社区支持体系。Python 显著降低了算法实现与实验验证的技术门槛，使得研究人员与开发者能够高效地完成从理论建模到模型迭代的全流程，加速了研究成果向实际应用的转换。因此，在人工智能实践中，Python 不仅是一种编程工具，更是连接算法理论与工程实现的关键桥梁，亦是当前数据科学与机器学习领域不可或缺的主流技术基础。本节将系统阐述 Python 的基础语法要素。

3.2.1　Python 数据结构

若将程序语法视作描述操作的“动词”，则数据结构即为承载信息的“名词”。人工智能在本质上可被视为对数据进行处理与转换的艺术，而 Python 提供了一系列高效且易用的内置数据结构，用以有效地组织与操作数据。

1. 基本数据类型

Python 的基本数据类型是构建复杂数据结构的基础，主要包括以下 4 类。

- **整数(int)**。表示无小数点的数值(如 5、−3)，支持算术运算(加减乘除、幂运算等)。
- **浮点数(float)**。表示带小数点的数值(如 3.14、2.0)，支持浮点运算，但浮点数在计算机中以二进制近似存储，导致某些十进制小数无法精确表示(如 0.1+0.2≠0.3)。

- **布尔值(bool)**。仅包含 True 和 False,用于逻辑判断(如 if 语句)。
- **字符串(str)**。由字符组成的不可变序列(如"hello"),支持拼接、切片和格式化操作。

Python 采用动态类型系统,无须在变量使用前显式声明其类型,变量的类型在赋值时被自动推断与绑定。这一特性既增强了代码的简洁性,也提升了编程的灵活性。

```
#Python 会自动推断变量类型
data_count = 1000                    #整数 (int)
learning_rate = 0.001                #浮点数 (float)
model_name = "ResNet"                #字符串 (str)
is_training = True                   #布尔值 (bool)
```

【例 3.3】 字符串及切片运算,针对字符串 S1="Guangzhou",如表 3.8 所示,提取出其中的元音字母。

表 3.8 字符串"Guangzhou"的正负向索引

字　符	G	u	a	n	g	z	h	o	u
正向索引	0	1	2	3	4	5	6	7	8
负向索引	−9	−8	−7	−6	−5	−4	−3	−2	−1

具体的切片操作如表 3.9 所示。

表 3.9 切片操作

元音字符	切片操作
u	S1[1]或者 S1[−8]
a	S1[2]或者 S1[−7]
ou	S1[7:]或者 S1[−2:]或者 S1[7:9]

2. 列表

列表(List)是一个有序且可变的数据集合,允许存储任意类型的数据(如数字、字符串、其他列表等),元素可增、删、改、查,允许重复。列表非常适合用来存储一批样本、一个时间序列的数据或神经网络中的层,常见操作如表 3.10 所示。

表 3.10 列表的常见操作

类　别	操作名称	方法/函数	说　明	示　例
创建	创建列表	[]/list()	创建一个新列表	lst=[1,2,3] lst=list((1,2))
访问	索引访问	lst[i]	访问第 i 个元素(从 0 开始)	lst[0]→1 lst[−1]→3
	切片	lst[start: end]	获取子列表,不含 end	lst[1: 3]→[2,3] lst[−2:]→[2,3]
添加	尾部添加	append(x)	添加一个元素到列表尾部	lst.append(4)
	插入指定位置	insert(i,x)	在索引 i 处插入元素 x	lst.insert(1,100)

续表

类　别	操作名称	方法/函数	说　明	示　例
删除	删除指定值	remove(x)	删除第一个值为 x 的元素	lst.remove(2)
	删除并返回值	pop()/pop(i)	默认删除最后一个；也可指定索引	lst.pop() lst.pop(1)
	删除指定位置	Del lst[i]	删除索引 i/i—j 的元素	del lst[2] del lst[1：3]
	清空列表	clear()	清除所有元素	lst.clear()
查找	查找索引	index(x)	找到元素 x 的位置	lst.index(2)
	计数	count(x)	统计元素 x 的出现次数	lst.count(2)
排序/反转	升序排序	sort()	原地排序(修改原列表)	lst.sort()
	降序排序	sort（reverse ＝ True)	原地降序排序	lst. sort（reverse ＝ True)
	排序(返回新列表)	sorted(lst)	返回排序后的新列表	sorted(lst)
	反转	reverse()	原地反转顺序	lst.reverse()
	反转(返回迭代器)	reversed(lst)	返回一个反转的迭代器	list(reversed(lst))
拷贝与合并	浅拷贝	lst.copy()/[:]	拷贝列表	lst2＝lst.copy()
	合并	＋	合并两个列表	lst3＝lst1＋lst2
	重复	*	重复列表多次	lst2＝lst * 3
内置函数	长度	len(lst)	返回列表长度	len(lst)→5

【例 3.4】　表 3.11 是暨南大学网络空间安全专业、计算机科学与技术专业 2024 年在广东省的高考录取分数线，请查找网络空间安全专业的平均分和计算机科学与技术专业的最低排位，并增加一条数据："软件工程　20　615　620　616.67　13462"。

表 3.11　暨南大学网络空间安全专业、计算机科学与技术专业 2024 年在广东省的高考录取分数线

专　业	录取人数	最低分	最高分	平均分	最低排位
网络空间安全	35	610	619	613.87	16070
计算机科学与技术	32	616	626	620.15	12721

代码 3.1　暨南大学网络空间安全专业、计算机科学与技术专业 2024 年在广东省的高考录取分数线数据处理(例 3.4)

```
#数据部分:每个子列表代表一个专业的数据
#列表格式:[专业, 录取人数, 最低分, 最高分, 平均分, 最低排位]
jn_scores = [
    ["网络空间安全", 35, 610, 619, 613.87, 16070],
    ["计算机科学与技术", 32, 616, 626, 620.15, 12721]
]
#查找"网络空间安全"专业的平均分
```

```
for row in jn_scores:
    if row[0] == "网络空间安全":
        avg_score_cyber = row[4]
        break
print("网络空间安全专业的平均分:", avg_score_cyber)
#查找"计算机科学与技术"专业的最低排位
for row in jn_scores:
    if row[0] == "计算机科学与技术":
        min_rank_cs = row[5]
        break
print("计算机科学与技术专业的最低排位:", min_rank_cs)
#增加一条专业数据:"软件工程 20 615 620 616.67 13462"
jn_scores.append(["软件工程", 20, 615, 620, 616.67, 13462])
#输出最新的数据列表
print("更新后的数据:")
for row in jn_scores:
    print(row)
```

运行结果：

```
网络空间安全专业的平均分: 613.87
计算机科学与技术专业的最低排位: 12721
更新后的数据:
['网络空间安全', 35, 610, 619, 613.87, 16070]
['计算机科学与技术', 32, 616, 626, 620.15, 12721]
['软件工程', 20, 615, 620, 616.67, 13462]
```

3. 元组

元组(Tuple)是一个有序但不可变的数据集合。一旦创建，其内容就不能被修改。这种“只读”属性使其非常适合存储那些不应被改变的数据，例如一个数据点的坐标、模型的固定配置参数等，元组的常见操作如表 3.12 所示。

表 3.12 元组的常见操作

类别	操作名称	方法/函数	说　明	示　例
创建	创建元组	() / tuple()	创建一个元组	t = (1, 2, 3) t = tuple([1,2])
	单元素元组	(x,)	注意逗号，单个元素也需加，	t = (5,)
访问	索引访问	t[i]	访问第 i 或第 i−j 个元素	t[0] → 1 / t[−1]→3 t[1:3] → (2, 3)
查询	是否存在	x in t/x not in t	判断元素是否存在	2 in t → True
	查找索引	t.index(x)	返回元素 x 的索引	t.index(2) → 1
	计数	t.count(x)	返回元素 x 出现的次数	t.count(2)
遍历	遍历元素	for x in t	遍历元组元素	for x in t:
	枚举遍历	enumerate(t)	同时获取索引和值	for i, v in enumerate(t):

续表

类别	操作名称	方法/函数	说　明	示　例
转换	元组转列表	list(t)	元组转换为列表，可变	list(t) → [1, 2, 3]
	列表转元组	tuple(lst)	列表转换为元组，不可变	tuple([1, 2]) → (1, 2)
合并/重复	合并元组	+	合并两个元组	t3 = t1 + t2
	重复元组	*	重复元组多次	t2 = t * 2→(1,2,3,1,2,3)
内置函数	长度	len(t)	返回元组长度	len(t) → 3
	最大/最小值	max(t) /min(t)	返回最值(元素可比较)	max(t) → 3
	求和	sum(t)	元素为数值时求和	sum(t) → 6
	并行遍历	zip(t1, t2)	多个元组打包成元组对	for a, b in zip(t1, t2):
嵌套	嵌套元组	(1,(2, 3))	元组可嵌套其他元组	t = (1, (2, 3))

【例 3.5】 根据例 3.4 中暨南大学网络空间安全专业、计算机科学与技术专业、软件工程专业 2024 年在广东省的高考录取分数线，设计元组存储各个专业的数据，编写代码输出各个专业录取的平均分。

代码 3.2　输出各个专业录取的平均分(例 3.5)

```
#用元组表示每个专业的录取数据
cyber_security = ("网络空间安全", 35, 610, 619, 613.87, 16070)
computer_science = ("计算机科学与技术", 32, 616, 626, 620.15, 12721)
software_engineering = ("软件工程", 20, 615, 620, 616.67, 13462)
#用列表存储所有专业的元组,便于批量处理
jn_admission = [cyber_security, computer_science, software_engineering]
#遍历所有专业,用元组解包,输出专业名和平均分
for name, quota, min_score, max_score, avg_score, enrollment in jn_admission:
    print(f"{name} 的平均分为{avg_score}")
```

4. 字典

字典(Dictionary)是键值对数据结构，其中键必须为不可变类型(例如字符串、整数或元组)，而值可为任意数据类型。通过键可实现对应值的高效存取；键必须是唯一且不可变的对象(通常为字符串或数字)。字典支持动态增删键值对，并具有极快的查询速度。它非常适合表示具有多个命名属性的对象，例如样本特征或模型超参数配置。字典的常见操作如表 3.13 所示。需要注意的是，在 Python 3.6 及之前版本中，字典为无序数据结构；而从 Python 3.7 开始，字典保持插入顺序。

【提醒】 字典在 Python 3.7 及以上版本有序(按插入顺序)、Python 3.6 及以下版本无序。

表 3.13　字典的常见操作

类别	操作名称	方法/函数	说　明	示　例
创建	创建字典	{} / dict()	创建一个空字典或使用键值对	d = {'a': 1, 'b': 2} d = dict(a=1)
	快速创建字典	dict.fromkeys(keys, val)	根据键列表创建字典，默认值相同	dict.fromkeys(['a','b'],0)

续表

类别	操作名称	方法/函数	说　明	示　例
访问	通过键取值	d[key]	获取键对应值(不存在则报错)	d['a'] → 1
	安全取值	get(key, default)	不报错,找不到返回默认值	d.get('x', 0) → 0
修改	修改或新增键值对	d[key] = value	如果键存在则更新,否则新增	d['a'] = 10
删除	删除指定键	pop(key)	删除并返回键对应的值,键不存在时报错	d.pop('a')
	删除指定键	pop(key, default)	键不存在返回默认值	d.pop('x', 0) → 0
	删除并返回项	popitem()	删除并返回最后插入的键值对	k, v = d.popitem()
	删除指定键	del d[key]	删除某个键(不可用时会出现异常)	del d['a']
	清空字典	clear()	移除所有键值对	d.clear()
遍历	遍历键	for k in d:	遍历所有键	for k in d:
	遍历键值对	items()	返回(key, value)组成的元组	for k, v in d.items():
	遍历键	keys()	返回所有键的视图对象	list(d.keys())
	遍历值	values()	返回所有值的视图对象	list(d.values())
	遍历所有键值对	items()	返回所有项(key, value)	d.items()→[('a', 1), ...]
查询	是否包含键	key in d / not in	判断是否存在某键	'a' in d → True
拷贝	浅拷贝	copy()	返回一个浅拷贝	d2 = d.copy()
合并	更新字典	update(other_dict)	用另一个字典更新当前字典(可覆盖)	d.update({'c': 3})
转换	转换为列表	list(d)	返回键列表	list(d) → ['a', 'b']

【例 3.6】 根据例 3.4 中暨南大学网络空间安全专业、计算机科学与技术专业、软件工程专业 2024 年在广东省的高考录取分数线,设计字典存储各个专业的数据,编写代码输出 3 个专业的最低排位和最低录取分。

代码 3.3　输出 3 个专业的最低排位和最低录取分(例 3.6)

```
#数据结构:key 为专业名,value 为[录取人数, 最低分, 最高分, 平均分, 最低排位]
jn_admission = {
    "网络空间安全": [35, 610, 619, 613.87, 16070],
    "计算机科学与技术": [32, 616, 626, 620.15, 12721],
    "软件工程": [20, 615, 620, 616.67, 13462]
}
#遍历字典,输出 3 个专业的最低排位和最低分
for major, data in jn_admission.items():
    min_score = data[1]
    min_rank = data[4]
    print(f"{major} 的最低录取分:{min_score},最低排位:{min_rank}")
```

5. 集合

集合(Set)是一个无序且元素唯一的可变容器，它主要用于去重和成员资格测试，支持添加、删除元素操作。其元素必须是不可变类型(如整数、字符串、元组)，由于元素无序，所以不支持索引访问。集合的常见操作如表 3.14 所示。

表 3.14　集合的常见操作

类别	操作名称	方法/函数	说　明	示　例
创建	创建集合	{} / set()	空集合用 set()，非空可直接用花括号	s = {1, 2, 3} s = set([1, 2, 3])
添加	添加单个元素	add(x)	添加元素 x 到集合中	s.add(4)
	添加多个元素	update(iterable)	用可迭代对象更新集合	s.update([5, 6])
删除	删除指定元素	remove(x)	删除元素 x，不存在时报错	s.remove(2)
	安全删除元素	discard(x)	删除元素 x，不存在也不报错	s.discard(10)
	弹出任意元素	pop()	随机删除一个元素并返回	s.pop()
	清空集合	clear()	移除所有元素	s.clear()
查询	是否包含元素	x in s / x not in s	判断元素是否在集合中	3 in s → True
遍历	遍历集合元素	for x in s:	集合是可迭代的	for i in s:
拷贝	浅拷贝	copy()	返回集合的副本	s2 = s.copy()
集合运算	并集	s1 | s2	s2/s1.union(s2)`	所有元素的合集
	交集	s1 & s2 / s1.intersection(s2)	两集合共有的元素	s1 & s2
	差集	s1 − s2 / s1.difference(s2)	s1 有但 s2 没有的元素	s1 − s2
	对称差集	s1^s2/s1.symmetric_difference(s2)	只存在于一个集合中的元素	s1 ^ s2
	子集判断	s1<=s2/issubset()	判断 s1 是不是 s2 的子集	s1 <= s2 / s1.issubset(s2)
	超集判断	s1 >= s2 / issuperset()	判断 s1 是不是 s2 的超集	s1 >= s2 / s1.issuperset(s2)
	是否相交	isdisjoint()	判断两个集合是否无交集	s1.isdisjoint(s2)
转换	集合转列表	list(s)	转为列表(可排序)	list(s)
	集合转元组	tuple(s)	转为元组	tuple(s)
去重	列表去重	set(list)	用集合快速去重	list(set([1,1,2,3]))→[1,2,3]
生成	集合推导式	{x for x in iterable}	用类似列表推导式的语法生成集合	{x * *2 for x in range(5)}

【例 3.7】 暨南大学计算机相关领域中排名前三的热门专业依次为网络空间安全、计算机科学与技术、软件工程；中山大学计算机相关领域中排名前三的热门专业则包括计算机科学与技术、软件工程及人工智能。需设计一个集合结构以存储各高校的前三名专业数据，并

编写相应代码以实现对中山大学和暨南大学特色专业的处理。

代码 3.4　中山大学和暨南大学的特色专业(例 3.7)

```
#暨南大学计算机相关领域中排名前三的热门专业
jnu_majors = {"网络空间安全", "计算机科学与技术", "软件工程"}
#中山大学计算机相关领域中排名前三的热门专业
sysu_majors = {"计算机科学与技术", "软件工程", "人工智能"}

#暨南大学的特色专业(只在暨南大学排名前三的专业中出现)
jnu_unique = jnu_majors - sysu_majors
#中山大学的特色专业(只在中山大学排名前三的专业中出现)
sysu_unique = sysu_majors - jnu_majors
print("暨南大学的特色专业:", jnu_unique)        #网络空间安全
print("中山大学的特色专业:", sysu_unique)       #人工智能
```

综上所述,Python 语言中常用的数据结构具备不同的特性与应用场景。列表作为一种有序且可变的数据结构,适合存储与操作一组需要动态修改的数据序列;元组同样具有有序性,但其不可变的特性使其常用于存储不应被更改的数据集合;字典以键值对的形式组织数据,能够支持基于键的高效检索与映射关系维护;集合作为一种无序容器,其内部元素具有唯一性,因而主要用于数据去重及执行各类集合运算。上述数据类型的核心特性总结如表 3.15 所示。

表 3.15　Python 的数据类型对比

数据类型	可　变　性	有　序　性	适用场景
基本类型	不可变	—	存储单一数值或文本
列表	可变	有序	动态数据集合(如训练样本)
元组	不可变	有序	固定数据集合(如模型参数)
字典	可变	Python 3.7+有序	键值映射(如特征索引)
集合	可变	无序	去重与集合运算(如特征筛选)

3.2.2　Python 控制结构

Python 语言的程序执行核心依托传统的三大基本控制结构:顺序结构、选择结构、循环结构。异常处理结构是程序的错误处理机制,用于增强程序的健壮性。程序的基本框架由控制流所决定。我们通常采用条件语句(例如 if-elif-else 结构)进行决策判断,并利用循环语句(如 for 循环)处理数据集中的每个数据点。Python 中的 for 循环本质上是一种遍历操作,能够迭代任何可迭代对象。这一特性使其与处理批量数据及迭代训练模型的人工智能应用场景高度契合。

1. 选择语句

if 语句作为一种基础的决策控制结构,其功能在于判断特定条件是否成立。若条件成立,则执行相应代码块;否则执行其他代码块。if 语句的语法定义如下:

```
if_stmt ::= "if" assignment_expression ":" suite
            ("elif" assignment_expression ":" suite) *
            ["else" ":" suite]
```

其中，方括号括起来的是可选的，其中“elif”语句可以有多个。

【例 3.8】　编程计算广州的出租车费用，规则是：12 元(含 3 千米)，3 千米后每千米 2.6 元，超过 15 千米后每千米 3.9 元。

代码 3.5　广州的出租车车费计算(例 3.8)

```
#12 元(含 3 千米),3 千米后每千米 2.6 元,超过 15 千米后每千米 3.9 元
km = float(input("请输入乘坐的千米数:"))
if km <= 3:
    fee = 12
elif km <= 15:
    fee = 12 + (km - 3) * 2.6
else:
    fee = 12 + (15 - 3) * 2.6 + (km - 15) * 3.9
print("应付车费:%.2f 元" % fee)
```

2. 循环语句

循环结构是程序设计中的基本控制结构，用于重复执行特定代码块。Python 语言主要提供了两种循环语句：while 循环与 for 循环。二者均可通过 break 语句提前终止循环，或通过 continue 语句跳过当前迭代，从而实现对循环流程的精细控制。

- while 循环适合在条件未知、需要不断尝试直到满足某个条件时使用。
- for 循环适合遍历序列(如列表、字符串、范围等)，语法简洁高效，常用于已知范围的重复操作。

与其他程序设计语言不同的是，Python 的循环语句可以有 else 子句，for…else 和 while…else 均支持“循环正常结束后执行 else”逻辑。

【例 3.9】　用 Python 编程判断一个数是不是快乐数(Happy Number)，即判断一个数不断把数字平方后求和，最终收敛到 1 称为“快乐数”。快乐数的定义为：对于一个正整数，不断将其各个数字的平方相加，经过若干次迭代后，若最终结果收敛为 1，则该数为快乐数；否则，该数不是快乐数。

代码 3.6　用 while 和 for 循环判断一个数是不是快乐数(例 3.9)

while 循环实现

```
n = int(input("请输入一个正整数:")) #快乐数有 1 7 10 13 19 23 28 31…
seen = set()
while n != 1 and n not in seen:
    seen.add(n)
    n = sum(int(digit) * * 2 for digit in str(n))
if n == 1:
    print("这是一个快乐数!")
else:
    print("这不是一个快乐数。")
```

for 循环实现

```
n = int(input("请输入一个正整数:"))
seen = set()
for _ in range(100):                                    #最多尝试 100 次,避免死循环
    if n == 1:
        print("这是一个快乐数!")
        break
    if n in seen:
        print("这不是一个快乐数。")
        break
    seen.add(n)
    n = sum(int(digit) * * 2 for digit in str(n))
else:
    print("未能判断是否为快乐数。")
```

3. 异常处理

异常处理机制旨在捕获并处理程序运行过程中出现的错误,以确保程序不会因错误而崩溃。常见的异常类型包括键值不存在、输入类型错误等。通过 try…except 结构,可以增强程序的健壮性并提升用户体验。在人工智能任务中,常需处理外部资源或非规范化数据,此类操作可能因多种原因失败(如文件不存在、数据格式错误)。采用 try…except 结构能够捕获这些异常情况,并执行备用方案,从而避免整个程序因单一错误数据而崩溃,确保代码的鲁棒性。

【例 3.10】 假设每个学生拥有一个个位数的幸运数字,并以字典形式存储。编写程序以查询学生的幸运数字。

代码 3.7 幸运数字的查询(例 3.10)

```
students = {"小明": 7, "小红": 3, "小刚": 9 }
try:
    name = input("请输入要查询的学生姓名:")
    print(name + "的幸运数字是:", students[name])
except KeyError:
    print("查无此人,请检查姓名是否正确。")
```

3.2.3 Python 函数

随着程序逻辑复杂度的提升,将代码组织为可复用的函数与模块,已成为提升开发效率与代码可维护性的关键途径。

在人工智能编程实践中,绝大多数操作均被封装于函数之中,例如数据预处理函数、模型构建函数、训练函数等。定义函数的核心在于明确其输入(参数)与输出(返回值),从而实现功能的抽象与封装。

1. 内置函数

函数是经过组织、可重复调用的代码单元,其实现了特定功能的模块化封装。Python 提供了丰富的内置函数,涵盖输入输出、类型转换、序列操作、数学运算及数据处理等方面,如表 3.16 所示。

表 3.16　Python 中常用的内置函数

分　类	常用函数	作用说明	示例代码及输出
输入输出	print()	输出信息	print("Hello, AI!") → Hello, AI!
类型转换	type()	查看类型	type(123) → <class 'int'>
	int()	转为整数	int("42") → 42
	float()	转为浮点数	float("3.14") → 3.14
	str()	转为字符串	str(100) → '100'
序列操作	len()	序列长度	len("abc") → 3
	range()	生成整数序列	list(range(3)) →[0, 1, 2]
	list()	转为列表	list("abc") → ['a','b','c']
数学运算	sum()	求和	sum([2, 4, 8]) → 14
	max()	最大值	max([2, 4, 8])→ 8
	min()	最小值	min([2, 4, 8]) →2
	abs()	绝对值	abs(−7) → 7
数据处理	sorted()	排序	sorted([8, 2, 4]) →[2, 4, 8]
	filter()	过滤数据	list(filter(str.isdigit, "a1b2")) → ['1','2']
	map()	映射处理	list(map(str.upper, "abc")) → ['A','B','C']

2. 自定义函数

自定义函数能够将重复的逻辑封装起来，从而显著提升代码的复用性与可维护性。自定义函数的基本语法结构如下：

```
def 函数名(参数 1, 参数 2=默认值):
    #此部分为函数的文档字符串(docstring),用于阐述函数的功能
    return 返回值
```

(1) 位置参数：在函数调用过程中，参数依据定义的顺序进行传递。

(2) 默认参数：作为可选参数，其具备预设的默认值。

(3) 返回值：通过 return 语句返回函数的执行结果。

【例 3.11】　使用 Python 编程输出 1000 以内的所有快乐数。为此，需定义一个自定义函数以判断给定数值是否为快乐数。快乐数是指通过迭代地将各位数字平方后求和，最终收敛为 1 的整数。

代码 3.8　输出 1000 以内的快乐数(例 3.11)

```
def is_happy_number(n):
    """
    判断一个数是否为快乐数。
    快乐数是指:对一个正整数不断将其各位数字的平方和替换原来的数,
    最终能得到 1 的数。如果陷入循环无法得到 1,则不是快乐数。
```

```
    """
    seen = set()
    while n != 1 and n not in seen:
        seen.add(n)
        n = sum(int(digit) * * 2 for digit in str(n))
    return n == 1
print("1000以内的快乐数有:")
for i in range(1, 1000):
    if is_happy_number(i):
        print(i, end=' ')
```

3. Lambda 函数

Lambda 函数亦称为匿名函数，是一种用于简洁地表达单行函数的语法结构。Lambda 函数常与高阶函数如 map()、filter()和 sorted()等结合使用，适用于快速定义简单的操作。其基本语法如下：

```
lambda 参数1, 参数2,… : 表达式
```

例如，定义 Lambda 函数 add = lambda x, y: x + y，执行输出运算 print(add(2, 3))后，结果为 5。

【例 3.12】 使用 Python 编程将给定的分数列表转换为及格/不及格标签，其中 scores = [55, 78, 90, 62, 45]。

代码 3.9 用 Lambda 函数将给定的分数列表转换为及格/不及格标签（例 3.12）

```
scores = [55, 78, 90, 62, 45]
labels = list(map(lambda x: "及格" if x >= 60 else "不及格", scores))
score_dict = dict(zip(scores, labels))
print("分数与标签的对应关系:", score_dict)
#结果:分数与标签的对应关系: {55: '不及格', 78: '及格', 90: '及格', 62: '及格', 45: '不及格'}
```

【例 3.13】 对模型的预测结果按置信度排序，results = [{"id": 1, "confidence": 0.82}, {"id": 2, "confidence": 0.95}, {"id": 3, "confidence": 0.67}]。

代码 3.10 用 Lambda 函数对模型的预测结果按置信度排序（例 3.13）

```
results = [
    {"id": 1, "confidence": 0.82},
    {"id": 2, "confidence": 0.95},
    {"id": 3, "confidence": 0.67}
]
sorted_results = sorted(results, key=lambda x: x["confidence"], reverse=True)
print("按置信度排序后的结果:", sorted_results)
#按置信度排序后的结果:
#[{'id': 2, 'confidence': 0.95}, {'id': 1, 'confidence': 0.82}, {'id': 3, 'confidence': 0.67}]
```