新工科建设·人工智能与智能科学系列教材

机器学习算法与实现

——Python 编程与应用实例

布树辉　李 霓　马文科　李永波　唐小军　张伟伟　编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

机器学习是人工智能的重要方向之一，对提升各行业的智能化程度正在起越来越大的作用。本书通过凝练机器学习的核心思想与方法，综合介绍了 Python、常用库和相关工具，以及机器学习的原理与实现，囊括了机器学习与行业相结合的实例，可让没有深厚计算机、编程背景的读者在有限的时间内掌握机器学习的相关知识和应用工具。本书各部分的比例适当，在讲授基本 Python 编程、库函数的基础上，由浅入深地介绍了机器学习的思想、方法和实现。理论讲授部分从基本的最小二乘法开始，逐步深入地介绍了如何使用迭代求解的方法实现逻辑斯蒂回归、感知机、神经网络、深度神经网络。本书配套有完整的在线讲义、在线视频、作业和练习项目，每章的习题、练习、报告等都配有对应的二维码，读者可直接访问在线教程，选择适合自己的资料。

本书可作为计算机、智能科学与技术、航空航天、电子信息、自动化等专业硕士研究生和本科生的教材，也可供相关技术人员参考。

图书在版编目（CIP）数据

机器学习算法与实现：Python 编程与应用实例/布树辉等编著．—北京：电子工业出版社，2022.12
ISBN 978-7-121-38633-6

Ⅰ．①机…　Ⅱ．①布…　Ⅲ．①机器学习－高等学校－教材　Ⅳ．①TN911.72

中国版本图书馆 CIP 数据核字（2022）第 036266 号

责任编辑：谭海平
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：21.5　　字数：578 千字
版　　次：2022 年 12 月第 1 版
印　　次：2022 年 12 月第 1 次印刷
定　　价：89.00 元（全彩）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）88254552，tan02@phei.com.cn。

序　言

深度学习作为当下人工智能方向的热点技术之一，推动了人工智能对产业应用的赋能。伴随着大数据、深度学习、智能芯片等技术的成熟，人工智能不仅在传统的图像、语音、自然语言领域得到了广泛而深入的应用，在物理、生物、化学等传统学科领域也深刻地改变着既有的技术模式。放眼世界，人工智能正成为国际竞争的新焦点。目前，人工智能相关领域的专业人员短缺，培养具备专业人工智能技能的人才是教育界面临的巨大挑战。

深度学习等机器学习方法有着较高的学习门槛，目前适合各学科的教材相对匮乏。国外此前出版的一些经典教材正面临着内容老化的问题，国内现有的大部分教材则侧重于介绍机器学习的理论，对读者在概率论、线性代数、计算机科学等方面的要求较高，读者学完相应的内容后依旧难以应用机器学习方法解决相关的难题。

本书最大的特点是，层层深入地通过实例来讲解机器学习的理论与算法，即以算法、代码、应用为主进行讲解，同时辅以详尽的算法解释、图解及代码运行结果，力求让读者在看懂代码的基础上深入理解算法，进而熟练使用算法；既可让读者高效、全面地掌握机器学习的主流知识点和整体脉络，又可让读者在遇到具体问题时方便查阅相关内容。

另一个特点是，在语言精确性和条理性、理论全面性和实践指导方面，集专业性与通俗性为一体。全书内容丰富，不仅包含 Python 语言、Python 第三方库、监督学习、无监督学习等机器学习内容，而且包含 sklearn、PyTorch、深度学习、目标检测、深度强化学习等前沿机器学习技术，既可作为高年级本科生和硕士研究生的教材，又可作为相关领域科研和工程人员的工具书。

人工智能赋能多个行业，大部分行业需要了解并掌握机器学习技术。开卷有益，希望本书能够辅助读者更好地掌握机器学习的理论、方法和应用。相信阅读此书所获得的充实感会给读者留下美好的回忆。

丁贵广
清华大学软件学院
2022 年 4 月 24 日

前　言

机器学习是人工智能领域的一个基础且重要的方向，它突破了传统手动设计处理算法所面临的通用性低、无学习能力等问题，通过设计学习模型让机器在大量数据中自动地挖掘出数据内在的规律，并对新数据进行预测或分类。最近十多年来，机器学习飞速发展，不仅在图像、视频、语音、自然语言等传统机器学习应用领域取得了进步，而且逐渐应用于航空航天、材料、化学、生物等诸多领域，为各学科开辟了新的数据处理与技术途径，并且产生了巨大的经济和社会效益。

全书共 11 章，综合介绍了 Python 编程、库函数等机器学习常用工具，由浅入深讲解了机器学习的核心思想与方法，给出了机器学习与航空航天等专业相结合的应用实例。为了让读者更好地掌握机器学习的相关知识，从 Python 编程、Python 常用库开始介绍；理论部分的讲解从基本的最小二乘法开始，逐步深入介绍如何使用迭代求解法实现逻辑斯蒂回归、感知机、神经网络、深度神经网络；每个理论知识点之后给出算法和编程实现，目的是帮助读者加深对理论的理解。本书配套有完整的在线讲义、在线视频、作业和练习项目、MOOC 等，并且每章都提供习题或练习等。对应的二维码如图 0.1 所示，读者可以直接访问在线教程，选择适合自己的资料。

(a)课程的配套视频　　(b)课程的在线讲义　　(c)作业和练习项目

图 0.1　配套资源的二维码

本书面向没有深厚计算机、编程基础的学生，考虑到不少读者并不具备深厚的编程和算法基础，在讲解过程中，更注重算法、数据结构、面向对象思想的引入，以引导读者通过“迁移学习”，将基础知识、专业理论、领域问题等快速迁移到机器学习领域，进而快速掌握机器学习的核心思想、方法和工具等，并为将机器学习技术应用到各自的专业领域以解决实际应用问题打下坚实的理论、编程和实践基础。

本书的第 1 章由布树辉和李霓编写，第 2、4 章由李霓编写，第 3 章由马文科编写，第 5 章由李永波编写，第 6 章由唐小军和张伟伟编写，第 7～11 章由布树辉编写。全书由布树辉统稿。

感谢为本书出版做出贡献的所有人员。

感谢多年来选择修习编著者团队的“智能图像处理”“机器学习”“机器学习与人工智能”课程的学生，是你们的积极、热情和反馈激励了编著者不断地改进讲义。

感谢西北工业大学飞行器智能认知与控制实验室的胡劲松、韩鹏程、李随城、胡博妮、陈霖、何学敏、李坤、邹万勇、左奎军、赵勇、薛少诚、夏震宇、董逸飞、王禹、翁乐安、张玉、乔岳、程佳铭、朱永宁、李俊峰、彭雨菲、王煜熙、倪海鸿、张敏、李浩玮、张一竹、贾旋等同学不辞辛苦地补充和改进讲义与书稿。

感谢深圳朝闻道机器人的张宏磊、陈豪杰、安玉玺等补充和协助。

感谢支持和帮助编著者的同事、朋友、亲人，有了你们的支持，本书才得以完成。

机器学习是一门范围极广、内容庞杂的学科，由于技术发展日新月异，编著者水平与经验有限，书中难免有错误与理解不到位的地方，敬请读者批评指正！

目　录

第1章 绪 论

人类长久以来的梦想是构造智能机器，让机器具备和人一样的学习能力，进而让机器更多、更好地完成任务。要让机器具备智能，学习的理论和方法就是重要的研究课题。在讨论机器如何学习之前，不妨先分析一下人类是如何学习的。人类的学习过程可以分为三个阶段：输入、整合和输出。例如，如果你要设计无人机系统，那么入门时就要学习飞机的基本结构，这就是输入阶段；然而，你很快就会发现自己不知道如何合理地组织飞机的各个部件，使其符合结构力学、空气动力学等基本力学原理，因此必须学习飞机的基本原理和力学相关知识，才能将这些结构最优地组织起来，这就是整合阶段；最后，根据你的设计与组织，一架飞机才能被设计出来，接着就可根据经验设计出各种各样的飞机系统，这就是输出阶段。

学习其他内容时，情形也是类似的，也就是说，都要经历经验积累、规律总结和灵活运用三个阶段。因此，我们可以对人类的学习给出如下定义：人类的学习是根据过往的知识或经验，对一类问题形成某种认识或总结出一定的规律，然后利用这些知识对新问题进行判断、处理的过程。人类虽然学习能力强，但是存在记忆能力弱、反应慢且容易出错等问题；计算机虽然不能像人一样灵活，但是容量大、计算速度快、运行稳定。如果能够充分结合二者的优势，使计算机以类似人类的方式解决更多复杂且多变的问题，就可产生可观的效益，进而在民用、军事等领域发挥巨大的价值。

如何更好地管理海量信息，挖掘所蕴含的知识，突破人类认知能力的瓶颈，是人类目前面临的主要挑战之一。机器学习（Machine Learning）便是这一理念的产物，因此可以给出机器学习的基本定义如下：机器学习是一类算法的总称，这些算法能够从大量历史数据中挖掘出隐含的规律，并用于分类、回归和聚类。机器学习更具体的定义是寻找从输入样本数据到期望输出结果之间的映射函数。注意，机器学习的目标是使学习得到的函数很好地适用于“新样本”[①]，而不是仅适用于训练样本。机器学习的基本思路如下：①收集并整理数据；②将现实问题抽象成数学问题；③利用机器学习求解数学问题；④评估求解得到的数学模型，如图 1.1 所示。无论使用什么算法和什么数据，机器学习的应用都包含上述几个步骤。

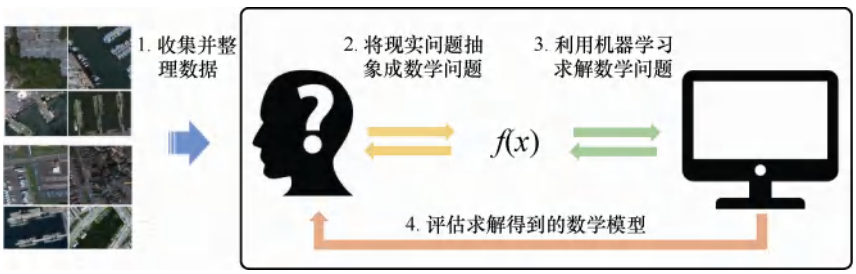


图 1.1 机器学习的基本思路

① 学到的函数适用于新样本的能力称为泛化（Generalization）能力。

1.1 机器学习的发展历程

人工智能（Artificial Intelligence，AI）是研究、开发用于模拟、延伸和扩展人类智能的理论、方法、技术及应用系统的技术科学，其目的就是让机器能像人一样思考，让机器拥有智能。从 20 世纪 50 年代至今，人工智能的发展经历了“推理期”“知识期”和“机器学习时期”。在推理期时代（1950—1970），研究人员将逻辑推理能力赋予机器，使其获得智能能力，虽然当时的 AI 程序能够证明一些著名的数学定理，但由于机器缺乏对知识的理解和整合，所以远不能实现真正的智能。

20 世纪 70 年代，人工智能的发展进入知识期（1970—1980），即研究人员通过总结与提炼人类的知识，以一定的模式教给机器，指导其完成某些复杂的任务，使机器获得一定程度上的应用智能。在这一时期，大量专家系统问世，并在很多领域取得成果，但由于人类知识量巨大，无法通过手工方式构建所有的知识。

由此可见，在人工智能的推理期和知识期，机器基本上是按照人类设定的规则和总结的知识运作的，不仅要耗费较高的人力成本，而且智能程度无法超越其创造者。于是，“机器的自我学习”需求慢慢凸显，此时大量基于机器学习的智能方法应运而生，成为实现人工智能的有力技术支撑，有限的规则和知识问题迎刃而解，人工智能进入机器学习时期（1980 年至今）。

机器学习时期大致也分为三个阶段。20 世纪 80 年代初，连接主义较为流行，代表性方法有感知机（Perceptron）和神经网络（Neural Network，NN）。20 世纪 90 年代，统计学习方法开始占据主流舞台，代表性方法有支持向量机（Support Vector Machine，SVM）。进入 21 世纪，深度神经网络被提出，连接主义思想又受到大家的关注；随着数据量和硬件计算能力的不断提升，以及互联网、物联网、5G 等技术的快速普及，以深度学习（Deep Learning，DL）为基础的诸多 AI 应用出现在大众视野，人工智能进入全新的时代。

图 1.2 显示了人工智能及机器学习发展的大致历程。

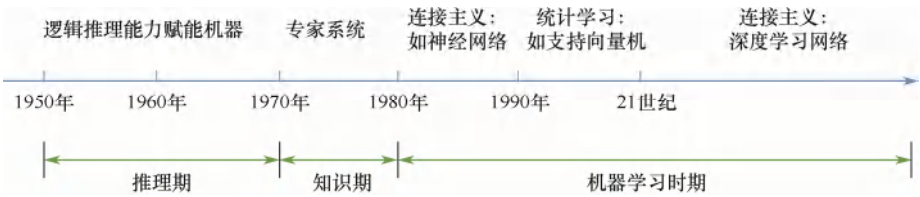


图 1.2 人工智能及机器学习发展的大致历程

1.2 机器学习的基本术语

要让机器学会认识世界，首先就要将现实世界的事物转换为数据表达，然后通过算法完成对数据的分析与判断。这个过程主要涉及如何表达现实世界的事物以及设计什么方法来进行判断等。为了更好地讲解机器学习的各个技术环节，下面简要介绍机器学习中的基本术语。

注意：若不太理解本章的内容，则不必强求理解，只需先记住有这些名词和概念即可，后续章节中将深入讲解涉及的概念与内涵。

1.2.1　特征

在机器学习中，特征（Feature）是被观测目标的一个独立可观测的属性或特点，其主要特点是存在信息量、区别性和独立性。特征通过一个抽象、简化的数学概念来代表复杂的事物。例如，识别草莓是否甜时，需要考虑的特征（属性）有尺寸、颜色、成熟度等；又如，识别某人时，可以结合其长相、走路姿势、说话声音等进行判断。为便于计算机处理和分析，特征常用数值而非文字或其他形式表示。特征和模型示例如图 1.3 所示。

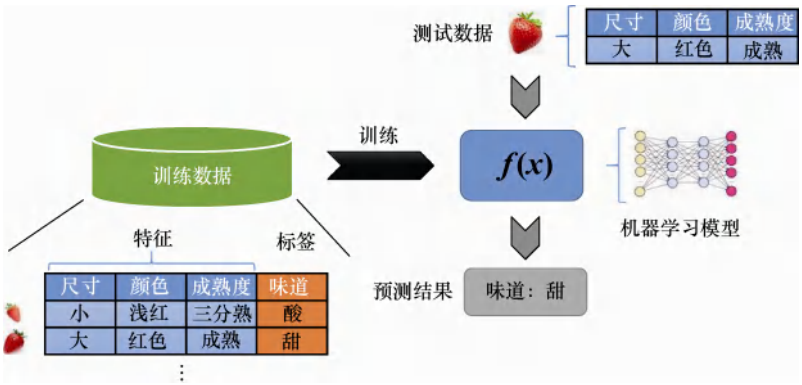


图 1.3　特征和模型示例

一个特征不足以代表一个目标，因此机器学习中使用特征的组合——特征向量来代表某个目标。例如，识别草莓是否甜时，使用三个维度的向量（尺寸、颜色、成熟度）来代表草莓。特征向量是为了解决实际问题而专门设计的。例如，进行人脸识别和水果种类识别时所用的特征向量肯定是不一样的。

1.2.2　样本

样本是数据的特定实例。样本分为两类：有标签样本和无标签样本。有标签样本同时包含特征和标签；而无标签样本只包含特征，需要算法根据数据在特征空间中的分布关系自动找到样本之间的关系。

样本的集合构成数据集。为了在不同的阶段对算法模型进行训练、测试和验证，数据集被分为训练集、测试集、验证集。训练集用于训练模型；验证集是模型训练过程中单独留出的样本集，用于调整模型的超参数[①]及初步评估模型的能力，常在模型迭代训练时用于验证当前模型的泛化能力（准确率、召回率等），进而决定是停止训练还是继续训练；测试集用来评估最终模型的泛化能力，但不能作为调参、选择特征等算法的依据。

1.2.3　模型

模型定义特征与标签之间的关系，一般可以视为一个由参数定义的函数或逻辑操作的集合。例如，草莓酸甜识别模型会将草莓大小、颜色、成熟度特征与酸/甜紧密联系起来。模型的生命周期包

① 在机器学习上下文中，超参数是在开始学习过程之前就设置了值的参数，而不是通过训练得到的参数。通常情况下，需要对超参数进行优化，为学习模型选择一组最优超参数，以提高学习的性能和效果。

括如下两个阶段。

- 训练阶段：创建或学习模型。向模型展示有标签样本，让模型逐渐学习特征与标签之间的关系。
- 推理阶段：将训练后的模型应用于无标签样本，使用经过训练的模型做出有用的预测。

1.2.4 回归、分类与聚类

机器学习的基本应用是回归（Regression）、分类（Classification）和聚类（Clustering）。回归和分类都属于监督学习，其主要特点是根据输入的特征，分析并得到数值或类别。回归问题常用于预测一个连续值，如预测房价、未来的气温等。比较常见的回归算法是线性回归（Linear Regression，LR）和支持向量回归（Support Vector Regression，SVR）。分类问题的目的是将事物打上标签，结果通常为离散值。例如，判断一张照片上的动物是猫还是狗时，分类的最后一层通常要使用 softmax 函数[①]来判断其所属的类别。常见的分类方法包括支持向量机、朴素贝叶斯（Naive Bayes）和决策树（Decision Tree）等。聚类问题属于无监督学习，用于在数据中寻找隐藏的模式或分组。聚类算法构成分组或类别，类别中的数据具有更大的相似度。聚类建模的相似度可以由欧几里得（欧氏）距离、概率距离或其他指标定义。常见的聚类算法包括 k 均值聚类、层次聚类（Hierarchical Clustering）和高斯混合模型（Gaussian Mixture Model，GMM）等。

1.2.5 泛化与过拟合

通过数据学习得到模型的过程就是学习（Learning），也称训练（Training）。在学习过程中，首先根据训练数据集构建一个模型，然后将该模型用于此前从未见过的新数据的过程，被称为模型的泛化（Generalization）。如果一个模型能够对从未见过的数据做出准确预测，就说明它能够从训练集泛化到测试集。在测试集上的评估是判断一个算法在新数据上表现好坏的一种方法。一般来说，简单模型对新数据的泛化能力更好。

构建一个对现有信息量来说过于复杂的模型后，如果该模型在拟合训练数据集时表现得非常好，但在测试数据集上的表现非常差，就说明模型出现了过拟合（Overfitting）问题。与之相反，模型如果过于简单，就可能无法掌握数据的全部内容及数据中的变化，机器学习模型甚至在训练集上的表现都很差，不能拟合复杂的数据，这称为欠拟合（Underfitting）。一般来说，模型越复杂，在训练数据上的预测结果就越好；然而，如果模型过于复杂，且过多地关注训练集中的单独数据点甚至异常点，模型就不能很好地泛化到新数据。模型复杂度与精度的折中如图 1.4 所示，即在训练集和测试集的精度都较高的情况下，才认为模型对数据拟合

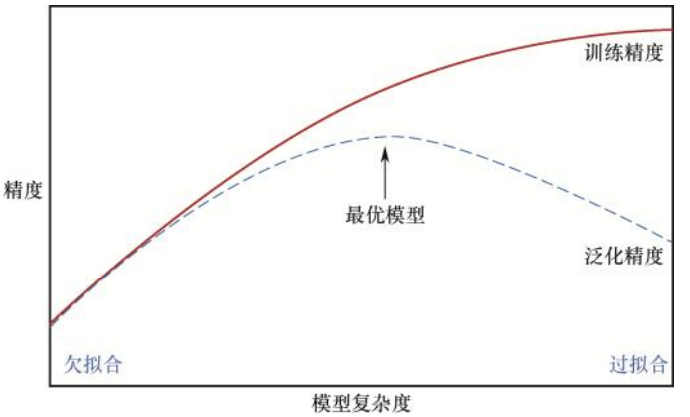


图 1.4 模型复杂度与精度的折中

① 又称归一化指数函数，目的是将多个分类的结果以概率的形式展现出来。

的程度刚刚好，模型的泛化表现出色，这才是最期望的模型。

1.3　机器学习的基本分类

机器学习建立模型并以求解模型参数的方式进行学习。下面按照模型的类别简要说明常用的机器学习方法。

1.3.1　监督学习

监督学习（Supervised Learning）是指在训练机器学习模型时，使用有标签的训练样本数据，首先建立数据样本特征和已知结果之间的联系来求解最优模型参数，然后通过模型和求解的模型参数对新数据进行结果预测。

监督学习通常用于分类和回归问题。例如，电子邮箱识别垃圾邮件的过程如下：首先对一些历史邮件做垃圾分类标记，接着对这些带有标记的数据进行模型训练，然后在获得新短信或新邮件时进行模型匹配，识别邮件是否是垃圾邮件。因此，电子邮箱识别垃圾邮件的过程就是监督学习下的分类预测。又如，航拍图像中的目标识别过程如下：对带有标记的图像数据进行模型训练，当新航拍图像被输入训练好的模型时，模型自动定位图像中目标的位置（回归），给出对应位置的类别（分类）。

监督学习的难点是，获取具有目标值的标签样本数据的成本较高，而成本高的原因是这些训练集要依赖于人工进行标注。监督学习的常见算法有感知机（Perceptron）、多层感知机（MultiLayer Perceptron，MLP）、支持向量机、逻辑斯蒂回归（Logistic Regression，LR）和卷积神经网络（Convolutional Neural Network）等。

1.3.2　无监督学习

在现实生活中，人们常常面临这样的问题：缺乏足够的先验知识，难以人工标注类别，或者人工标注类别的成本太高。自然地，在人工提供少量帮助或者不提供帮助时，我们希望计算机能够自动完成数据的内在规律学习。根据类别未知（未被标记）的训练样本解决模式识别中的各种问题，称为无监督学习（Unsupervised Learning）。

无监督学习与监督学习的区别是选取的样本数据无须标签信息，而只需分析这些数据的内在规律。无监督学习常用于聚类分析，如客户分群，即通过客户的消费行为（消费次数、最近消费时间、消费金额）指标对客户数据进行聚类和分析。此外，无监督学习也适用于降维。无监督学习相对于监督学习的优点是，数据不需要进行人工标注，且数据获取成本低。

常用的无监督学习算法主要有聚类、主成分分析（Principal Component Analysis，PCA）、等距映射、局部线性嵌入等。

1.3.3　半监督学习

在现实生活中，无标签的数据易于获取，而有标签的数据收集起来通常较为困难，标注起来也耗时耗力。在这种情况下，半监督学习（Semi-Supervised Learning）更适合于现实世界中的应用，近年来已成为深度学习领域的热门方向。半监督学习是监督学习和无监督学习相结合的一种学习方法，半监督学习方法可以结合分类、回归和聚类，只需少量有带标签的样本和大量无标签的样本即可获得较好的应用效果。

- 半监督分类是指在无标签样本的帮助下训练有标签样本，获得比只用有标签样本训练时更优的分类。
- 半监督回归是指在有标签样本的帮助下训练无标签样本，获得比只用有标签样本训练时更好的回归。
- 半监督聚类是指约束聚类的数据集中还包含一些关于聚类的监督信息，常见的约束有必须连接（must-link）约束和不能连接（cannot-link）约束，分别表示两个样本点一定在一个类别中或者一定在不同的类别中。约束聚类的目标就是提升无监督聚类的表现。
- 半监督降维是指在有标签样本的帮助下，找到高维输入数据的低维结构，同时保持原始高维数据和成对约束的结构不变。

半监督学习是最近比较流行的方法。在算法层面上，半监督学习介于监督学习和无监督学习之间，包括对一些常用监督学习算法的延伸，这些算法试图对未标注数据建模，然后对已标注数据进行预测。隐藏在半监督学习下的基本规律是，数据的分布必然不是完全随机的，通过一些有标签数据的局部特征及更多无标签数据的整体分布，可以得到能够接受甚至非常好的分类结果，如半监督支持向量机（Semi-Supervised Support Vector Machine，S3VM）等。

1.3.4 深度学习

根据人工神经网络[①]的层数或者机器学习方法的非线性处理深度，可以将机器学习算法分为浅层学习算法和深度学习算法。浅层学习算法的典型代表是感知机，深度神经网络主要是指处理层数较多的神经网络。深度学习是机器学习领域的一个新研究方向，将深度学习引入机器学习的目的是使后者更接近最初的目标——人工智能。深度学习的灵感源于人类大脑的工作方式，是利用深度神经网络来解决特征表达的一种学习过程。一个用于图像差异检测的深度神经网络示例如图 1.5 所示，该网络（Shape-Aware Siamese Convolutional Network，SASCNet）由差异特征提取模块、差异融合模块和融合微调模块构成，且这些模块是基于不同网络结构针对不同图像特性设计的。多层网络设计可以更好地挖掘图像特征，进而提高对目标差异的检测率。

深度学习是一类模式分析方法的统称，就具体的研究内容而言，主要涉及三类方法。

- 基于卷积运算的神经网络系统，即卷积神经网络。
- 基于多层神经元的自编码神经网络，包括自编码（Auto Encoder）及近年来受到广泛关注的稀疏编码（Sparse Coding）。
- 以多层自编码神经网络的方式进行预训练，进而结合鉴别信息进一步优化神经网络权重的深度置信网络（Deep Belief Network，DBN）。

通过多层处理，逐渐将“低层”特征表示转换为“高层”特征表示后，再用“简单模型”即可完成复杂的分类、回归等学习任务。由此，可以将深度学习理解为特征学习（Feature Learning）或表征学习（Representation Learning）。以往在将机器学习用于现实任务时，描述样本的特征通常需要由人类专家来设计，这称为特征工程（Feature Engineering）。众所周知，特征的好坏对泛化性

① 人工神经网络（Artificial Neural Networks，ANNs），也称神经网络，是一种模仿动物神经网络行为，进行分布式并行信息处理的网状机器学习模型，它通过调整内部大量节点之间相互连接的关系，达到处理信息的目的。

能有着至关重要的影响，而人类专家设计出好的特征并非易事；特征学习（表征学习）则通过机器学习技术自身来产生好的特征，这就使得机器学习向“全自动数据分析”前进了一步。

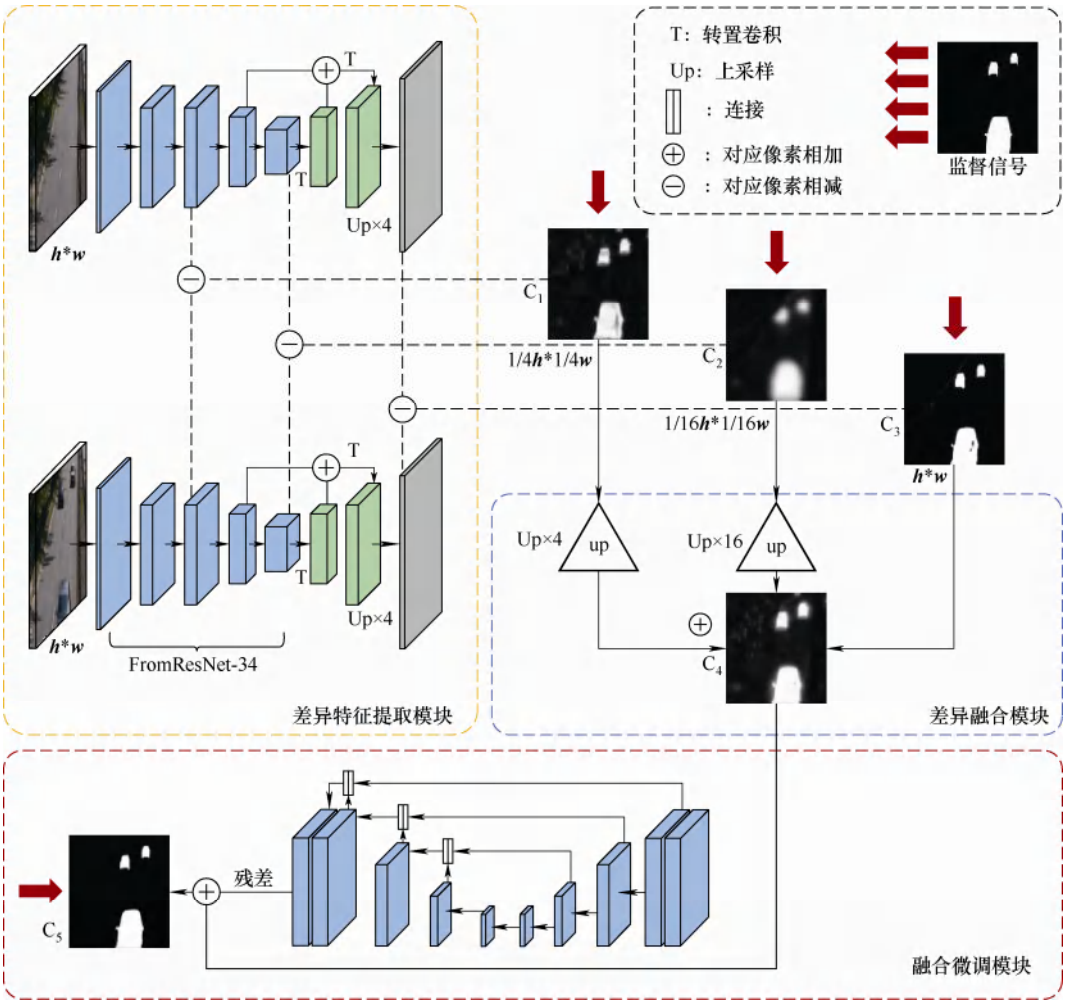


图 1.5　一个用于图像差异检测的深度神经网络示例

浅层学习算法主要对一些结构化数据、半结构化数据场景进行预测，而深度学习算法主要解决复杂的场景问题，如图像、文本、语音识别与分析等，具体体现如下。

- 强调了模型结构的深度，通常有 5 层以上的隐藏层节点。
- 明确了特征学习的重要性。也就是说，通过逐层特征变换，将样本在原空间的特征表示变换到一个新特征空间中，从而使分类或预测更容易。与人工规则构造特征的方法相比，利用大数据来学习特征更能够表征数据丰富的内在信息。

通过设计建立适量的神经元计算节点和多层运算层次结构，选择合适的输入层和输出层，通过网络的学习和调优，建立从输入到输出的函数关系，虽然不能百分之百地找到输入与输出的函数关系，但是可以尽可能地逼近现实的关联关系。使用训练成功的网络模型，就可满足我们对复杂事务处理的自动化要求。

常见的深度学习模型包括卷积神经网络模型、深度置信网络模型和堆栈自编码网络（Stacked Auto-Encoder Network）模型等。对于不同的网络和任务数据，采用自下而上的无监督学习或自顶向下的监督学习优化方式对网络权重参数进行优化。

1.3.5 强化学习

强化学习（Reinforcement Learning，RL）又称再励学习、评价学习或增强学习，用于描述和解决智能体与环境交互的学习策略，以达成回报最大化或者实现特定目标的问题，强化学习示意图如图 1.6 所示。强化学习虽然需要监督信息，但是这种监督信息要在若干动作序列之后才能得到，因此与监督学习所需的明显因果关系标签不完全一样。强化学习强调系统与外界之间的不断交互反馈，主要针对智能体和环境交互过程中不断需要推理的场景，如无人汽车驾驶。

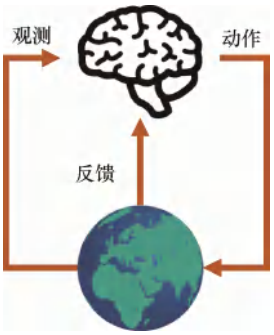


图 1.6 强化学习示意图

强化学习理论受到行为主义心理学启发，侧重于在线学习并试图在“探索”和“利用”之间保持平衡。不同于监督学习和无监督学习，强化学习不要求预先给定任何数据，而通过接收环境对动作的奖励（反馈）来获得学习信息并更新模型参数。强化学习解决的问题具有如下特点：①智能体和环境之间不断进行交互；②搜索和试错；③延迟奖励，当前所做的动作可能要在多步之后才会产生相应的结果。强化学习是机器学习大家族中的重要分支，由于近年来自身理论的发展及与深度学习的整合，强化学习得到了快速发展和广泛应用。例如，在 AlphaGo 成功挑战世界围棋高手、控制战斗机进行空中格斗、精准预测蛋白结构等方面，强化学习都取得了巨大的成功。

强化学习的常见模型是标准的马尔可夫决策过程（Markov Decision Process，MDP）。按给定条件，强化学习可以分为基于模式的强化学习（Model-based RL）和无模式强化学习（Model-free RL），以及主动强化学习（Active RL）和被动强化学习（Passive RL）。强化学习的变体包括逆向强化学习、阶层强化学习和部分可观测系统的强化学习。求解强化学习问题所用的算法可分为策略搜索算法和值函数（Value Function）算法两类。深度学习模型可在强化学习中得到使用，形成深度强化学习（Deep Reinforcement Learning，DRL）。深度强化学习带来的推理能力真正让机器拥有了自我学习能力，这也是衡量机器智能的一个关键指标。深度强化学习本质上属于采用神经网络作为值函数估计器的一类方法，其主要优势是能够利用深度神经网络自动抽取状态特征，避免了人工定义状态特征带来的不准确性，使得智能体能够在更原始的状态上进行学习。

1.3.6 机器学习与人工智能

“人工智能”最初是在 1956 年的达特茅斯学会上提出的。自此，世界范围内的研究人员发展了众多的相关理论，人工智能的概念得以充实，这个具有前瞻性的学科也随之发展起来。人工智能的主要目标是使机器能够胜任通常人类智能才能完成的复杂工作。人工智能是以计算机科学为基础，融合了心理学、哲学、电子信息科学的交叉学科，研究对象主要包括机器人、语音识别、图像识别、自然语言处理和专家系统等。

机器学习是一种实现人工智能的方法，是一种实现机器学习的技术，也是当前最热门的机器学习方法。图 1.7 显示了机器学习和人工智能之间的关系。由图可知，人工智能出现得最早，是最大、最外侧的同心圆，因此是一个最广泛的概念。实现人工智能的各类技术或方法不是独立存在的，而

是相互交叉而有一定交集的。监督学习、半监督学习、无监督学习、强化学习主要从训练数据是否包含标注信息来进行区分；深度学习主要从非线性处理层的深度来划分。深度学习所提供的基础模型可以应用到监督学习、半监督学习、无监督学习、强化学习等各个领域，是提升机器学习性能的有力支撑。

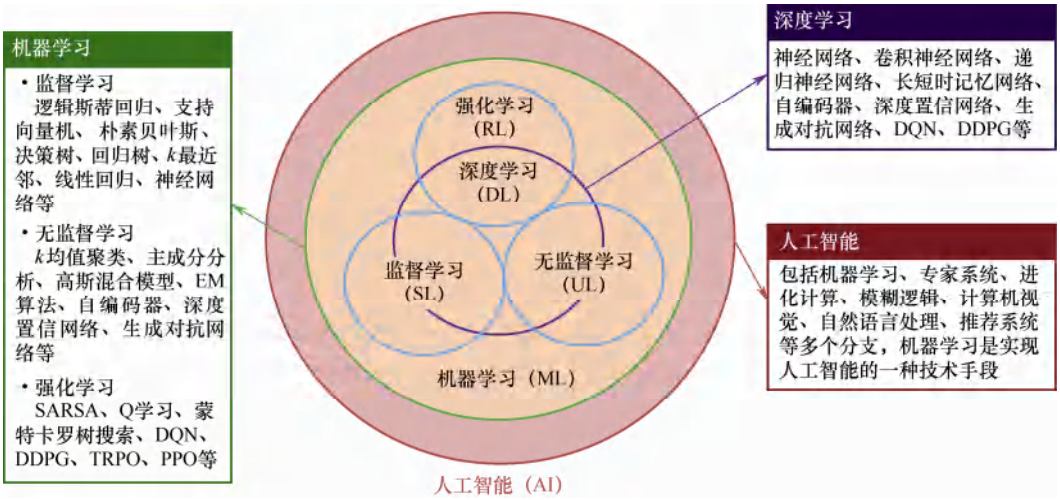


图 1.7　机器学习和人工智能之间的关系

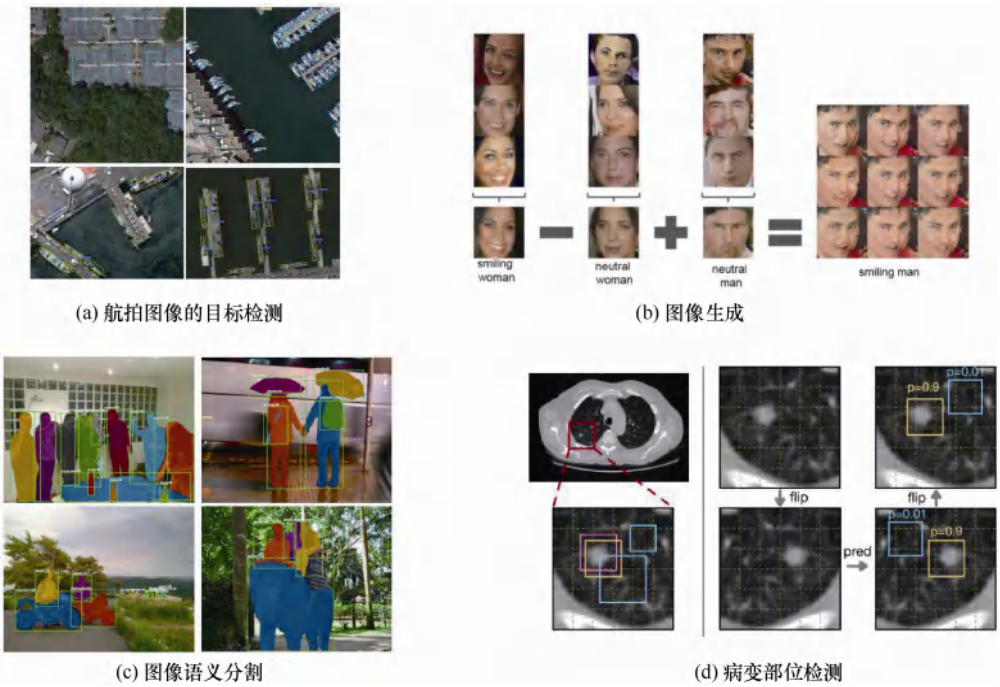
1.4　机器学习的应用

以机器学习为代表的各种智能方法不仅应用于图像识别、语音识别、自然语言处理等传统领域，在环境感知、智能决策等领域也应用广泛，近年来还开始延伸到了气动、强度、结构设计等领域，极大地优化了设计效率和效果。此外，机器学习技术还延伸到了化学、物理、生物等学科的实验数据处理、知识发现方面，极大地拓宽了人类的认知边界。按照目前的技术发展态势，机器学习会更多地应用于各行各业的各项任务中，并会带来翻天覆地的变化。由于机器学习的应用领域较多，本节简要梳理并分析几类比较重要的应用，以帮助读者认识机器学习的重要性。

1.4.1　图像识别与处理

图像识别与处理是最重要的机器学习应用之一，它通过提取图像的特征，实现自动分类、分割、识别、编辑、增强等操作，提高部署应用的自动化程度，提升机器效率。图像识别与处理的部分应用示例如图 1.8 所示。图像识别主要包括三个阶段——识别文字信息、识别数字化信息和识别目标。图像识别经过这三个阶段的发展，充分发挥自身的特点与优势，逐步拓展到各个领域，并与各行业的技术相结合。图像识别技术的主要应用与发展方向有字符识别、机器视觉识别、生物医学等。

在航空领域，图像识别应用于军事侦察、目标识别、场景识别等；在交通领域，图像识别应用于道路识别、车辆车牌检测等；在安防领域，基于图像识别技术的视频智能分析系统能够实现人脸识别、人脸支付、智能自动化监控等；在医学领域，微创手术中的手术导航技术运用图像识别技术，在心脏、脑结构等器官的病变部位的辅助识别方面有着不可替代的作用；在农业领域，图像识别技术在病虫害诊断、检测农作物生长等方面发挥着巨大的作用。



(a) 航拍图像的目标检测　(b) 图像生成

(c) 图像语义分割　(d) 病变部位检测

图 1.8　图像识别与处理的部分应用示例

1.4.2　语音识别与自然语言处理

语音识别能够将语音识别为文本，可以实现人工智能助理。例如，Cortana 是微软公司研发的一款典型人工智能助理，它伴随 Windows 10 操作系统的发布而推出；苹果公司的软件中也推出了 Siri。使用这些产品，可以极大地方便人们的生活。

自然语言处理（Natural Language Processing，NLP）是计算机科学领域与人工智能领域的一个重要研究方向，主要研究在人与计算机之间使用自然语言进行有效通信的各种理论和方法，主要目的有二：自然语言理解，即让计算机理解自然语言文本的意义；自然语言生成，即让计算机能以自然语言文本来表达给定的意图、思想等。近年来，伴随着机器学习技术的快速发展，自然语言处理领域取得了很大的进展。各种词表、语义语法词典、语料库等数据资源的日益丰富，词语切分、词性标注、句法分析等技术的快速进步，各种新理论、新方法、新模型的出现，推动了自然语言处理研究的长足发展。

语音识别是指将一段语音信号转换为相对应的文本信息，本质上是一种基于语音特征参数的模式识别。通过事先准备好的语料和模型进行学习，语音识别系统能将输入的语音按一定的模式进行分类，进而依据判定准则找出最佳匹配结果。随着深度学习技术的发展，深度神经网络已经替代了传统的高斯混合模型、支持向量机等，且相比传统方法的优势如下：使用深度神经网络估计的后验概率分布不需要对语音数据分布进行假设；深度神经网络的输入特征可以是多种特征的融合，包括离散的特征或连续的特征；深度神经网络可以利用相邻语音帧中包含的结构信息进行特征提取，进而增强语言特征表达能力。语音识别领域的应用主要包括语言翻译、阅读理解、个人助理、聊天机器人、知识图谱、高考机器人、调查分析等。

1.4.3　环境感知与智能决策

传统机器学习主要处理传感器采集的固定数据，如监控图像、指纹图像、语音等。随着机器人/无人机技术的发展，人们对智能体自主运行能力的要求越来越高，这就要求智能体能够在与环境交互的过程中对所采集的数据进行实时重建、识别、理解和决策。由于需要与环境实时交互，所以智能体面临着多方面的挑战：①智能体配置了多种传感器，如相机、激光雷达、惯性传感器、雷达、红外相机等，如何有效地融合多种不同类型的传感器？②智能体需要和环境交互，如何实时处理多种传感器的数据？③如何分析感知的数据，获得语义层面的信息和认知层面的知识等，进而实现“随机应变”的能力？图 1.9 显示了机器人和无人机环境感知与智能决策应用示例。

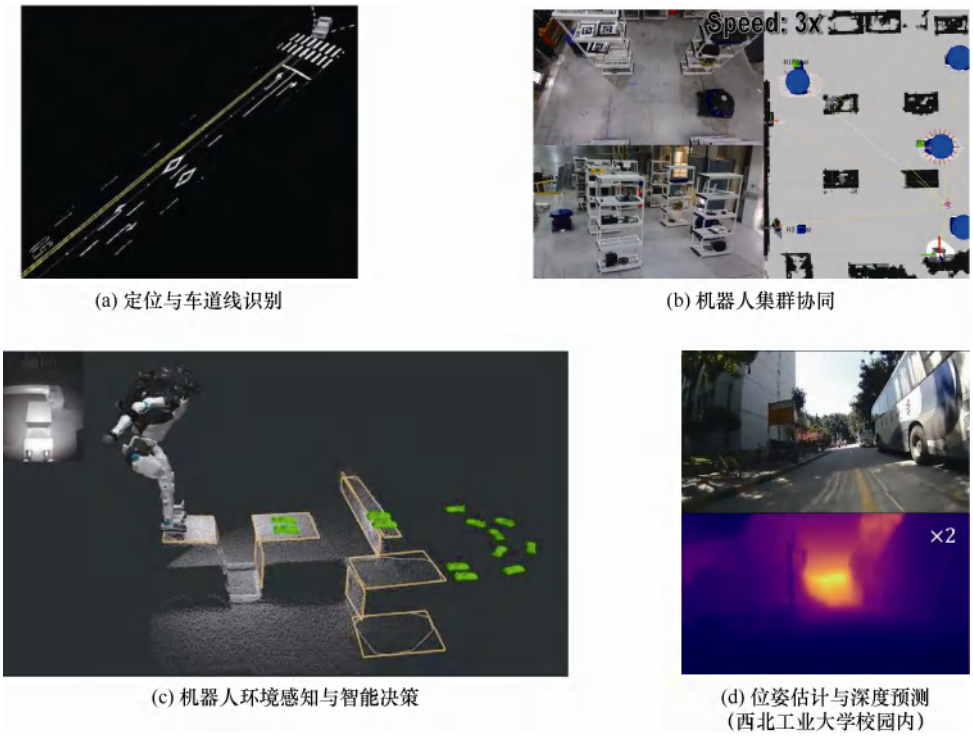


图 1.9　机器人和无人机环境感知与智能决策应用

环境感知一直是机器人/无人机领域的研究热点之一。环境感知主要包括定位和环境构建、目标识别、目标追踪、路径规划等。环境感知算法对不同的应用场景有着不同的研究重点。例如，在测绘、AR/VR 等领域，需要尽可能详细地展示实际环境的几何、色彩等特征细节，而对实时性要求不高；又如，在机器人/无人机运动规划领域，环境感知侧重于定位精度应尽可能高、地图占用的内存应尽可能小、地图重建的效率应尽可能高，在功能方面还需要标记出地图中的移动障碍物，并且测量移动障碍物的运动速度及其运动轨迹等。随着近年来深度学习技术的发展，通过深度学习方法引入的语义信息可以显著提高识别、追踪效果。此外，根据不同的硬件算力对网络进行优化，可以在精度与效率之间达到平衡。

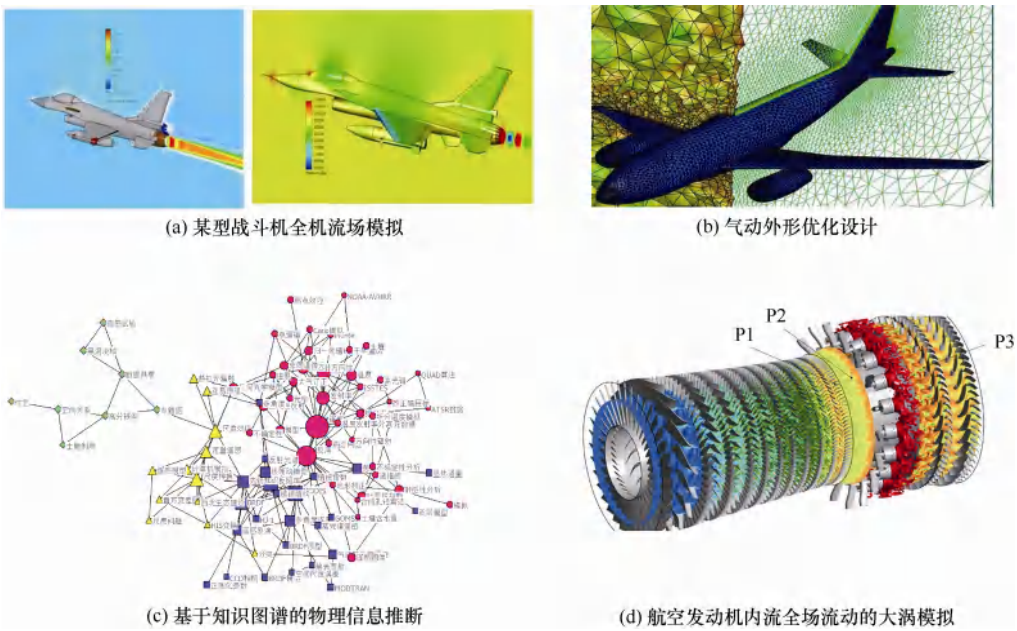
智能决策是指让智能体学会“选择”。智能体的决策控制系统的任务是，根据给定的先验信息和自身行驶状态，结合行为预测、路径规划和避障机制，自主产生合理运行的决策，实时完成智能体的动作

规划。这个过程面临的难题如下：①最优选项的确定；②选错的代价设计；③决策的连锁性，即一个决策对另一个决策的影响；④决策的正确性有时取决于另一个决策，即“博弈性”。智能决策的主要应用包括动作规划、障碍物规避、路径规划、集群控制、协同控制等。机器学习在这一系列智能感知和决策过程中起关键作用，是实现高阶机器智能的核心技术。

1.4.4 融合物理信息的工程设计

随着人工智能技术的飞速发展，以机器学习为代表的多种智能技术已被广泛应用于飞行器的气动、强度、结构设计等多个学科领域。首先，通过机器学习方法对多源异构的海量数据进行知识抽取、知识融合、知识推理和知识表达，可协助专家完成飞行器的设计、维护、任务规划以及更高层级的设计任务。航空飞行器的种类和状态繁多，但是数据仍然较少，利用针对小样本的迁移学习、元学习等技术，可以将训练好的模型更好地泛化到更多的状态和对象，快速、高效、稳定地迁移到新的学习、预测任务中。其次，基于知识图谱框架，通过融合专家知识、经验和数学模型，可使机器学习具备更好的可解释性、稳定性和可信任性。

在气动与多学科优化设计中，可采用机器学习方法开展基于集成学习、强化学习等手段的气动外形优化设计。对于高维设计问题，可从稀疏学习、特征分析的角度进行设计空间和设计分层分析，从而大幅度提升气动设计的效率。湍流、气动噪声和结构疲劳等是飞行器设计中长期面临的难题，随着人工智能时代的到来，机器学习方法为上述问题提供了新的解题思路。例如，针对湍流问题，基于机器学习的湍流模型为新一代湍流模型的构建提供了极大的自由度，为工程湍流模型的定制提供了方法基础。又如，基于数据驱动的机器学习方法可用于飞行器气动噪声源的智能识别、噪声强度的智能预测、噪声诱发机理及降噪措施的研究等方面。再如，在飞行器结构强度分析中，利用机器学习在数据挖掘等方面的优势，融合不确定性表达与推理、跨域数据的多模态特征提取与融合、小样本知识迁移等技术，可提高结构疲劳寿命预测的准确性。图 1.10 显示了融合物理信息的工程设计相关应用。



(a) 某型战斗机全机流场模拟　(b) 气动外形优化设计

(c) 基于知识图谱的物理信息推断　(d) 航空发动机内流全场流动的大涡模拟

图 1.10　融合物理信息的工程设计相关应用

1.5　机器学习应用的步骤

使用机器学习解决实际问题时，遵循的基本流程如下。

- 选择合适的模型。这通常需要视实际问题而定，即要针对不同的问题和任务选取恰当的模型，而模型就是一组函数的集合。
- 判断函数的好坏。这需要确定一个衡量标准，即通常所说的损失函数（Loss Function），损失函数的确定也需要视具体问题而定，如回归问题一般采用欧氏距离，分类问题一般采用交叉熵代价函数。
- 找出“最好”的函数的模型参数。如何从高维参数空间最快地找出最优的那组参数是最大的难点。常用的方法有梯度下降法、最小二乘法等。
- 学习得到“最好”的函数的参数后，需要在新样本上进行测试，只有在新样本上表现很好时，才算是一个最优模型。

机器学习算法的选择只是其中的步骤之一，其他步骤、数据、算法、算力等对机器学习也至关重要。机器学习的流程本质上是数据准备、数据分析、数据处理和结果反馈的过程，如图 1.11 所示。按照这一思路，可将机器学习分为如下步骤：应用场景分析，数据处理，特征工程，算法模型训练与评估，应用服务。

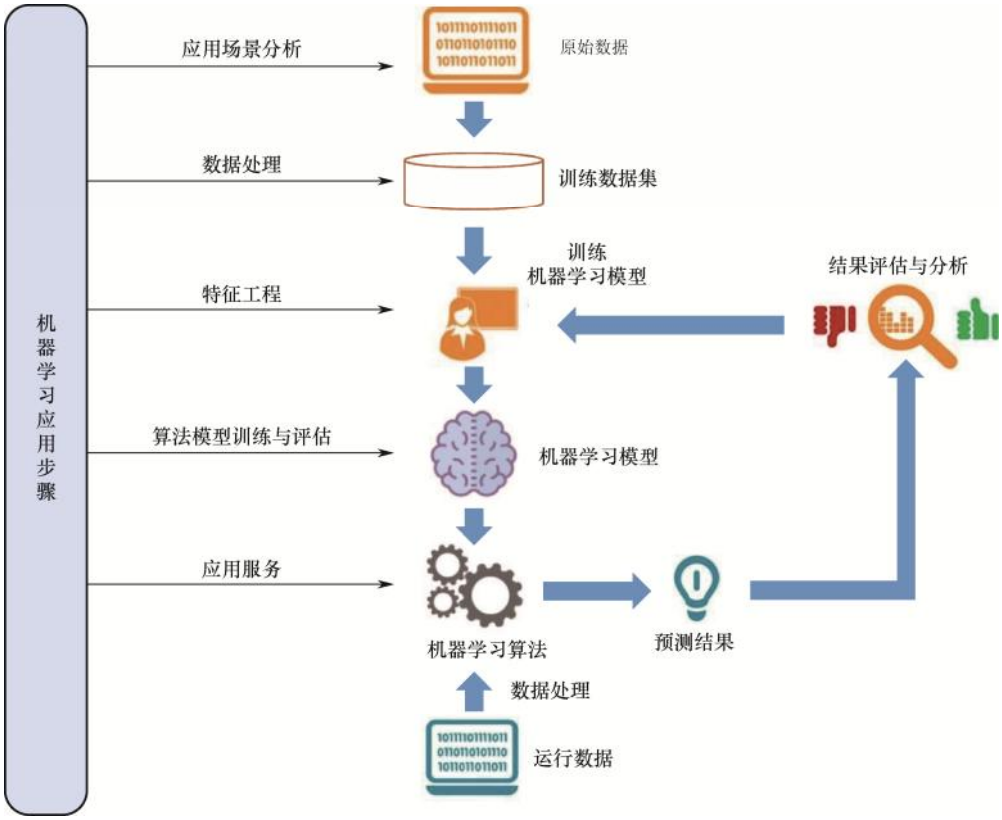


图 1.11　机器学习的流程

1.5.1 应用场景分析

应用场景分析是指将应用需求、使用场景转换为机器学习需求的语言，然后分析数据、选择算法的过程。这是机器学习的准备阶段，主要包括场景抽象、数据准备和算法选择。

（1）场景抽象。本质上，场景抽象是指针对目标需求，将所面临的问题抽象为应用场景的问题，而场景抽象是指将目标需求抽象为分类、聚类、回归等具体的机器学习问题。例如，现代化战场中，目标的动态检测与追踪就是一类目标识别分类问题。

（2）数据准备。机器学习的基础是数据，没有数据就无法训练模型。数据准备包括识别数据、收集数据和加工数据。通过不同应用渠道获取的数据有结构化数据、半结构化数据、非结构化数据。机器学习能够结合不同的方式和方法来处理这些数据。对于数据，需要考虑两个关键问题：①数据量，要求数据量尽可能大；②数据的缺失，要求尽可能地完善收集的数据。

（3）算法选择。算法选择根据需求、数据特性选择最优的算法模型。由于存在很多候选算法，所以要根据问题特性、性能、扩展性、实现的便捷性等多方面综合选择最优的算法。

1.5.2 数据处理

数据处理是指数据的选择和清洗过程，目的是尽可能地去除数据中的噪声等对算法的干扰，所用的主要方法包括去噪和归一化。一般情况下，采集的数据能够反映真实情况，存在传感器噪声、采集过程受到的干扰等，将这样的数据直接输入模型会影响算法的性能。因此，需要通过算法识别干扰数据。一般来说，归一化是指将数据的值域调整到区间$[0,1]$或$[1,-1]$，以帮助算法更快、更好地寻找最优解。输入的数据有多种量纲，如数据表示的时间可能以小时为单位，也可能以分钟为单位，混合使用多种量纲的数据必然导致很大的误差。因为数据分析过程不考虑量纲，所以需要对数据进行归一化处理，并统一数据的量纲。归一化的另一个作用是提高算法的收敛速度。例如，假设要分析 X 和 Y 的因果关系，X 的值域是$[1,10]$，Y 的值域是$[1,100000]$。由于两个数据的值域差别太大，算法的收敛速度将显著降低，所以需要对数据进行归一化处理。还有很多不同的方法和手段能够帮助我们解决数据处理过程中的问题，降低“坏”数据对模型的干扰。

1.5.3 特征工程

在机器学习领域，数据和特征决定了机器学习的上限，而模型和算法只能逼近这个上限，因此数据和特征是算法模型的基础。所谓特征工程，是指对处理完的数据进行特征提取，将其转换为算法模型可以使用的数据。

特征工程是指将原始数据转换为能够更好地表达问题本质的特征的过程。将这些特征应用到预测模型中，能够提高对不可见数据的模型预测精度。特征是对建模任务有用的属性，这就意味着并非所有属性都可视为特征，区分的关键是看该属性对解决问题有无影响。特征与属性的不同之处是，特征可以表达更多的与问题上下文有关的内容，且可被运算或计算。

特征工程的目的如下：从数据中抽取对预测结果有用的数据；从数据中构建对结果有用的信息；寻找更好的特征以提高算法效率；寻找更好的特征，达到选择简单模型就能取得更好拟合效果的目的。一般情况下，在数据处理过程中可以同步进行特征工程，如归一化处理。特征工程处理过程包括特征选择、特征提取和特征构造，是机器学习中的重要一环，特征的好坏直接影响最终算法的性能。同一组数据，使用相同的算法和不同的特征时，得到的结果会有很大的差别。

1.5.4　算法模型训练与评估

模型训练是指在数据准备、数据处理、特征工程之后，根据选取的算法对机器学习模型进行训练与评估。模型训练流程通常具有如下功能：在训练集上进行训练；在验证集上进行验证；模型可以保存和读取每次训练的参数，并且读取权重；记录训练集和验证集的精度，以便调整参数。在机器学习模型的训练过程中，模型只能利用训练数据进行训练，而不能接触测试集中的样本，因此模型很容易出现过拟合问题，所以需要在训练集中分割出一部分数据，构建不参与训练的验证集，用于评估模型在未知样本上的泛化能力。一般来说，机器学习模型有众多的网络结构和超参数，因此需要反复尝试，通过多次训练找到最优的网络结构和超参数配置。机器学习模型的精度与模型的复杂度、数据量、数据增广等因素直接相关，因此，当机器学习模型处于不同的阶段（欠拟合、过拟合和完美拟合）时，需要使用不同的方法和技巧来优化模型。

1.5.5　应用服务

机器学习应用服务是指模型训练完成后，如何部署机器学习、如何进行数据处理、如何输入模型和推理，以及如何快速训练模型、配置模型相关参数。模型应用可通过应用程序接口（Application Programming Interface，API）供应用层调用，应用层也可通过配置页面来配置模型的相关参数，如置信度等。为了更好地将机器学习作为服务提供给客户，一般需要设计并开发一个机器学习平台来让各个模块有机地协同，主要考虑要素包括：①模块分层。系统之间通常由来自不同团队的很多人维护，为了能够高效地迭代演进，系统之间需要保证足够的松耦合性。一个系统的内部实现机制的变化或者版本发布，都不应该对其他系统进行修改。另外，在特征处理、计算框架等部分需求比较多样化的环节，需要具有较好的可插拔能力。②代码化。复杂的业务逻辑往往需要众多系统配合，基于周期或条件判断运行。驱动这些系统的应是自动化的程序和配置，而不能仅仅是手动管理。③可观测。可观测是运维、优化和排错的基础，主要分为配置可观测和过程可观测。

1.6　机器学习的评估方法

当模型训练完成时，或者当模型进行上线前测试时，需要对模型进行跟踪与评估。机器学习的目的是使学习得到的模型能够很好地适用于新样本，即让模型具有泛化能力。然而，在训练数据上表现得很好的模型，在测试数据上不一定表现得好，原因是模型将训练数据的特有性质当作所有样本都具有的一般性质，导致泛化能力减弱，出现欠拟合问题。欠拟合常在模型学习能力较弱而数据复杂度较高的情况下出现，此时模型由于学习能力不足，无法学习到数据集中的一般规律，导致泛化能力减弱。

为了充分验证机器学习模型的性能和泛化能力，需要选择与问题相匹配的评估方法，才能发现模型选择或训练过程中的问题，进而迭代地优化模型。模型评估主要分为离线评估和在线评估。针对分类、回归、聚类等不同类型的机器学习问题，评估指标的选择是有所不同的。知道每种评估指标的精确定义，有针对性地选择合适的评估指标，根据评估指标的反馈进行模型调整，是模型评估阶段的关键问题。下面介绍机器学习评估中的数据集划分方法和性能度量方法。

1.6.1　数据集划分方法

数据集一般需要分割为训练集、测试集和验证集，划分方法如下。

1. 留出法

直接将数据集 D 划分为两个互斥的集合，即预留出一部分作为测试集和验证集，其他则作为训练集。训练集、测试集和验证集的划分要尽可能地保持数据分布的一致性，避免因数据划分过程引入额外的偏差而对最终结果产生影响。

- 优点：可以保持数据分布的一致性。
- 缺点：当训练集过大而测试集和验证集较小时，评估结果缺乏稳定性；当测试集和验证集过大时，将导致与根据训练集训练出的模型差距较大，缺乏保真性。

2. 交叉验证法

交叉验证（Cross Validation）法首先将数据集 D 划分为 k 个大小相似的互斥子集，然后用 $k-1$ 个子集的并集作为训练集，余下的子集作为测试集和验证集。这样，就得到 k 组训练集/测试集和验证集，进而进行 k 次训练、测试和验证，最终返回 k 个测试结果的均值。

- 优点：更稳定、更准确。
- 缺点：时间复杂度较大。

3. 自助法

首先对包含 m 个样本的数据集 D 进行随机采样，构建 D'（抽取 m 次），然后每次都将抽取的数据对象放回 D，D' 则用作训练集，剩余的数据则用作测试集。某样本不被抽取的概率为

$$\lim_{m\to+\infty}\left(1-\frac{1}{m}\right)^m \approx 0.368 \tag{1.1}$$

因此，初始样本集 D 中约有 73.2%的样本作为训练集，36.8%的样本作为测试集和验证集。

- 优点：适合较小且难以有效划分训练集/测试集的数据集。
- 缺点：产生的数据集会改变原始数据集的分布，会引入估计偏差。

1.6.2 性能度量

性能度量（Performance Measure）是衡量模型泛化能力的数值评价标准，反映了所求解模型的性能。使用不同的性能度量会导致不同的评价结果；模型的好坏不仅取决于算法和数据，而且取决于当前的任务需求。根据目标任务的不同，需要采用不同的性能度量。对于回归问题，性能度量方法采用均方误差，均方误差越小，模型效果越好；对于分类问题，评价分类器性能的指标一般是分类准确率（Accuracy），即对于给定的测试数据集，分类器正确分类的样本数与总样本数之比；对于二分类问题，常用的评价指标是精确率（Precision）与召回率（Recall）。通常以关注的类别为正类，其他类别为负类，分类器在测试数据集上的预测或者正确，或者不正确，四种情况出现的总数分别记为 TP（将正类别预测为正类别的数量，简称真正例）、FN（将正类别预测为负类别的数量，简称假反例）、FP（将负类别预测为正类别的数量，简称假正例）、TN（将负类别预测为负类别的数量，简称真反例），如表 1.1 所示。

表 1.1 模型预测的评价参数

真实情况	预测结果	
	正 例	反 例
正 例	TP（真正例）	FN（假反例）
反 例	FP（假正例）	TN（真反例）

精确率定义为

$$p=\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} \tag{1.2}$$

召回率定义为

$$R=\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \tag{1.3}$$

此外，还有 F_1 值，它是精确率和召回率的调和均值，即

$$\frac{2}{F_1}=\frac{1}{p}+\frac{1}{R} \tag{1.4}$$

精确率和召回率都高时，F_1 值也高。

1.7　如何学习机器学习

阅读前面的内容后，相信读者愿意领略机器学习的美妙。根据笔者多年的教学经验，真正学得较好的读者都掌握了一定的学习技巧，且花了较多的时间和精力。有的读者可能会担心自己的基础薄弱，不知道从何入手。不用担心！只要多动脑、勤动手，很快就可以入门。下面提供一些有助于提升学习效率的建议。

1.7.1　由浅入深

机器学习综合了高等数学、矩阵论、概率论、统计学、计算方法、算法、编程等诸多学科的知识，大量庞杂的知识会让初学者晕头转向。此外，机器学习的本质是研究一系列算法来分析数据，而非计算机专业的读者往往不具备计算机原理、编程等相关专业的知识，直接学习晦涩难懂的抽象概念和算法程序，往往只能学到表面的知识，而很难深刻理解底层的原理，因此不会灵活运用机器学习方法去解决实际问题。针对机器学习课程的特点及读者专业的特点，本书由浅入深、循序渐进地介绍机器学习核心的思想、方法和算法，并基于 Python 应用实例首先让读者建立感性认识，在学和做的过程中不断强化理解机器学习的深度与广度。

1.7.2　行成于思

人的时间和精力是有限的，即使最终理解并掌握了这门课程，但是如果花了太多的时间，就会耽误其他课程的学习。如果为了准备数学“装备”而从指数、对数、导数、矩阵、概率和统计等一一学起，就要花费很长的时间，这是不现实的。此外，在攀登机器学习这座“高山”时，如果要理解全部方法和概念，就可能因为追寻绝佳景色的道路过于漫长而中途放弃。因此，这里建议读者首先快速建立感性认识去解决一些实际问题，并在解决问题的过程中加深对理论、算法、编程、验证等方面的理解。此外，要尽可能在掌握基本知识后开始做习题和小项目，在做的过程中去查阅资料，不断完善自己的知识。要尽可能采用循环迭代的方式进行学习，不要强求自己一步就学会，而要在做与学的过程中不断强化感性认识，进而牵引理论学习。

机器学习是一门实践性很强的学科，需要理论与实践相结合。学好一门知识的最好办法是使用它，因此建议读者一定要自己动手进行实际操作，尽可能地实现书中的代码。本书自第 2 章起的各章都提供作业，此外还提供练习项目，要通过完成作业和练习项目来加深对所学知识的理解。有的读者看完数学公式后觉得自己理解了相关内容，而在编写程序时却不知道如何下手，这时编写程序就是验证自己是否真正理解相关内容的一种方法。

本书适用于希望攀登机器学习这座“高山”的读者，可以让读者在最短的时间内领略机器学习世界的绝佳景色。下面开启“机器学习”的学习旅程。

第 2 章　Python 语言

Python 是一种解释型的、高级的、通用的、面向对象的动态语言，主要特点包括：简单易用，学习成本低；拥有大量的标准库和第三方库；开源且免费使用。Python 和其他语言的不同之处是，不使用花括号等表示代码的层级关系，而使用代码缩进表示代码的层级关系。Python 最初设计用于编写自动化脚本，经过不断更新和迭代，Python 正越来越多地应用于独立的大型项目开发，并且在机器学习领域中得到了广泛和深入的应用。目前，大部分机器学习框架、库都是基于 Python 语言开发的，因此学习 Python 不仅可以帮助读者学习最新的机器学习技术，而且可为读者在各自的专业领域应用机器学习奠定编程基础。图 2.1 显示了本章配套资源的二维码。

(a)本章配套在线视频　　(b)本章配套在线讲义

图 2.1　本章配套资源的二维码

2.1　为什么选择 Python

最好的程序员不是为了得到更高的薪水或公众的仰慕而编程，他们只是觉得这是一件有趣的事情。

—— Linux 之父 Linus Torvalds

Python 的设计哲学是优雅、明确、简单，且与其他编程语言相比，初学者更容易上手。Python 的目标是提供简单且好用的解决方法，使用户能够专注于要解决的问题。其次，Python 功能强大，标准库或第三方库封装了很多功能，用户无须考虑底层的实现细节，只需调用即可。使用 Python 解决问题的特点如图 2.2 中的漫画所示。

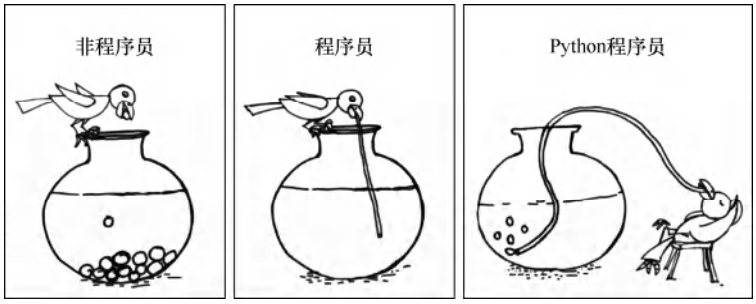


图 2.2　漫画《口渴的 Python 开发者》形容了 Python 开发者有多么轻松（摘自 Pycot 网站）

为什么要选择学习 Python？学会后可以用来做什么？Python 能做的事情很多，如能够应用到如下场景。

- 机器学习。Python 具有上手容易、易用等特点，具有丰富的机器学习框架，更多地关注网络架构和损失函数设计，底层的数值计算、优化等由机器学习库完成，因此能够快速地验证想法，提高机器学习研究、开发的速度。

- 数据分析。Python 快速开发的特性支持迅速实现验证数据处理、分析的想法，可避免开发人员将时间花费在程序编写、调试上，丰富的第三方库简单易用，能够加快开发、测试进度。
- 数据爬取。Python 可以快速处理网络数据，使用正则表达等方式迅速将网页中关注的信息解析出来，进而抓取想要的数据。
- 网站后端。使用 Python 搭建网站和后台服务时，比较容易开发和维护，且很容易添加新功能。借助功能丰富的网站框架 Django、Flask 等，可以快速搭建网站，还可以使移动端自适应。
- 自动化运维。随着集群系统的采用，人们越来越多地使用自动化手段来实现运维。Python 在系统运维方面的优势是其强大的开发能力和完整的工具链。
- 自动化测试。测试需要执行大量的重复操作和判断等，而 Python 提供了简单、易用的测试框架、工具，能够极大地加速测试的自动化。

Python 又称胶水语言，因为它能以混合编译的方式使用 C/C++/Java 等语言的库，从而弥补自身运行效率不高的问题。作为一门历史悠久的语言，Python 拥有良好的生态、较低的学习门槛、方便的开发体验，已成为大众最喜欢的语言之一。

本书中的示例程序由 Python 及第三方库实现。为了方便后面的学习，下面简要介绍 Python 语言的基础和特点。

2.2　安装 Python 的环境

由于 Python 的库较多，且依赖关系比较复杂，Python 一般都通过包管理系统来管理计算机中安装的各个 Python 软件包。目前，主要的包管理系统有 pip 和 Anaconda。为了让初学者快速地使用 Python，本书主要以 Anaconda 为例介绍如何安装。

注意：本书的安装说明中有较多的细节，可能与读者的系统不适配，因此可能会遇到问题。遇到问题时，请通过搜索引擎查找解决办法，以提高自己解决问题的能力。

2.2.1　Windows 下的安装

Anaconda 集成了大部分 Python 软件包，因此方便使用。由于跨境网络的下载速度较慢，推荐使用镜像来提高下载速度。

（1）在镜像网站[①]上找到适合自己的安装文件，然后下载。例如，

https://mirrors.bfsu.edu.cn/anaconda/archive/Anaconda3-2020.11-Windows-x86_64.exe

（2）按照安装提示与说明，安装 Anaconda。

2.2.2　Linux 下的安装

通过网站下载并安装最新的 Anaconda 安装文件，如代码 2.1 所示。

代码 2.1　下载与安装 Anaconda

```
01: # 下载
02: wget https://mirrors.bfsu.edu.cn/anaconda/archive/Anaconda3-2020.11-Linux-x86_64.sh
03:
04: # 安装
05: bash ./Anaconda3-2020.11-Linux-x86_64.sh
```

① https://mirrors.bfsu.edu.cn/anaconda/archive/。

按照提示逐步操作，完成安装。

注意：在安装过程中需要选择自动加入环境变量的选项。安装完成后要先关闭终端，然后打开终端即可使用。

2.2.3 设置软件源

1. 设置 Conda 软件源

参考 Conda 安装和软件源的设置说明[①]。在各种操作系统下都可修改用户目录下的.condarc 文件。Windows 用户无法直接创建名为.condarc 的文件，可执行 conda config –set show_channel_urls yes 生成该文件后再行修改。

打开文件编辑器，编辑“~/.condarc”。例如，在 Linux 中，可在终端执行“gedit ~/.condarc”，然后将下面的内容复制到该文件中，具体如代码 2.2 所示。

代码 2.2 .condarc 配置

```
channels:
  - defaults
show_channel_urls: true
default_channels:
  - https://mirrors.bfsu.edu.cn/anaconda/pkgs/main
  - https://mirrors.bfsu.edu.cn/anaconda/pkgs/r
  - https://mirrors.bfsu.edu.cn/anaconda/pkgs/msys2
custom_channels:
  conda-forge: https://mirrors.bfsu.edu.cn/anaconda/cloud
  msys2: https://mirrors.bfsu.edu.cn/anaconda/cloud
  bioconda: https://mirrors.bfsu.edu.cn/anaconda/cloud
  menpo: https://mirrors.bfsu.edu.cn/anaconda/cloud
  pytorch: https://mirrors.bfsu.edu.cn/anaconda/cloud
  pytorch-lts: https://mirrors.bfsu.edu.cn/anaconda/cloud
  simpleitk: https://mirrors.bfsu.edu.cn/anaconda/cloud
```

2. 设置 pip 源

继续输入如下命令设置 pip 源，如代码 2.3 所示。

代码 2.3 设置 pip 源

```
pip config set global.index-url 'https://mirrors.ustc.edu.cn/pypi/web/simple'
```

2.2.4 安装常用 Python 库

打开 Conda 的命令行程序，输入下面的命令，如代码 2.4 所示。

代码 2.4 安装软件

```
conda install jupyter scipy numpy sympy matplotlib pandas scikit-learn
```

2.2.5 安装 PyTorch

打开 Conda 的命令行程序，输入下面的命令，如代码 2.5 所示。

① https://mirrors.bfsu.edu.cn/help/anaconda。

代码 2.5　安装 torchvision 软件

```
conda install pytorch -c pytorch
pip3 install torchvision
```

2.2.6　Conda 使用技巧

1. 创建新的虚拟运行环境

使用如下命令生成一个新环境，如代码 2.6 所示。

代码 2.6　生成一个新环境

```
conda create -n <your_env>
```

注意：上面的<your_env>要替换成读者期望的虚拟环境名。

2. 激活虚拟运行环境

激活一个虚拟环境，如代码 2.7 所示。

代码 2.7　激活虚拟环境

```
conda activate <your_env>
```

3. Conda 常用命令

Conda 常用命令如代码 2.8 所示。

代码 2.8　Conda 常用命令

```
# 帮助命令
conda -h
conda help

# 退出当前环境
conda deactivate

# 查看基本信息
conda info
conda info -h

# 查看当前存在环境
conda env list
conda info --envs

# 删除环境
conda remove -n your_env --all
```

更多技巧请查阅网络上的资源[①]。

2.3　Jupyter Notebook

本书配套的示例程序是使用 Jupyter Notebook 编写的。使用 Jupyter Notebook，可以方便地将理论、公式、程序、数据可视化等集成在多媒体页面上，方便读者阅读和学习。

① https://blog.csdn.net/marsjhao/article/details/62884246。

Jupyter Notebook 是一种 Web 应用，它能让用户将说明文本、数学公式、代码和可视化内容组合到一个易于共享的文档中，非常方便教学与研究，让使用者一目了然。Jupyter Notebook 特别适合处理数据，如数据清理和探索、可视化、机器学习和大数据分析，具体特点如下。

- 编程时具有语法高亮、缩进、制表位补全功能。
- 可直接通过浏览器运行代码，同时在代码块下方显示运行结果。
- 以富媒体格式显示计算结果。富媒体格式包括 HTML、LaTeX、PNG、SVG 等；对代码编写说明文档或语句时，支持 Markdown 语法。
- 支持使用 LaTeX 编写数学性说明。

安装 Jupyter Notebook 最简单的方法是使用 Anaconda，其发行版中附带有 Jupyter Notebook。要在 Conda 环境下安装 Jupyter Notebook，可在终端输入命令 conda install jupyter，也可以通过 pip 来安装：pip install jupyter。安装后，就可在终端输入以下命令启动，如代码 2.9 所示。

代码 2.9 启动 Jupyter Notebook

```
01: # jupyter notebook
```

执行命令后，终端中将显示一系列 Jupyter Notebook 的服务器信息，同时浏览器自动启动 Jupyter Notebook。在启动过程中，终端显示的内容如代码 2.10 所示。

代码 2.10 启动时终端显示的内容

```
01: jupyter notebook
02: [I 08:58:24.417 NotebookApp] Serving notebooks from local directory:/Users/catherine
03: [I 08:58:24.417 NotebookApp] 0 active kernels
04: [I 08:58:24.417 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
05: [I 08:58:24.417 NotebookApp] Use Control-C to stop this server and shut down all kernels
    (twice to skip confirmation).
```

Jupyter Notebook 会在当前计算机的 8888 端口打开一个服务，通过访问 http://localhost: 8888 即可进入 Jupyter Notebook 的界面，其中 localhost 指的是本机，8888 指的是端口号。要想自定义端口号来启动 Jupyter Notebook，可在终端输入以下命令，如代码 2.11 所示。

代码 2.11 自定义端口号启动 Jupyter Notebook

```
01: # jupyter notebook --port <port_number>
```

其中，<port_number>是自定义端口号。

注意：使用 Jupyter Notebook 进行编写操作时，不要关闭终端。一旦关闭终端，就会断开与本地服务的连接，导致在 Jupyter Notebook 中无法进行其他操作。

2.3.1 Jupyter Notebook 的主页面

执行启动命令后，浏览器进入 Jupyter Notebook 的主页面，如图 2.3 所示。

打开 Jupyter Notebook 后，默认目录是执行命令时所在的目录。要想在指定目录下使用 Jupyter Notebook，只需事先使用切换目录命令（如 cd）切换到目标路径。要想新建一个 Jupyter Notebook，只需单击 New 按钮，然后选择想要新建的 Python 版本。新建一个标签页后，可以看到 Jupyter Notebook 界面，当前界面是空白的，如图 2.4 所示。

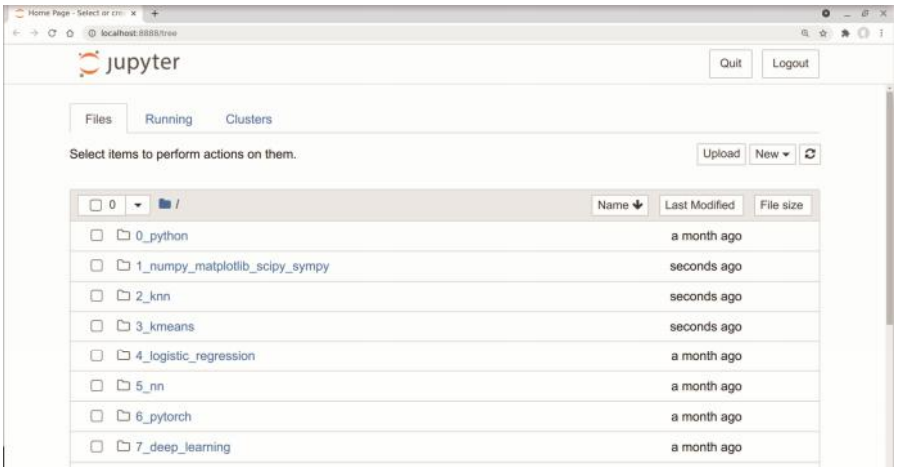


图 2.3　Jupyter Notebook 的主页面

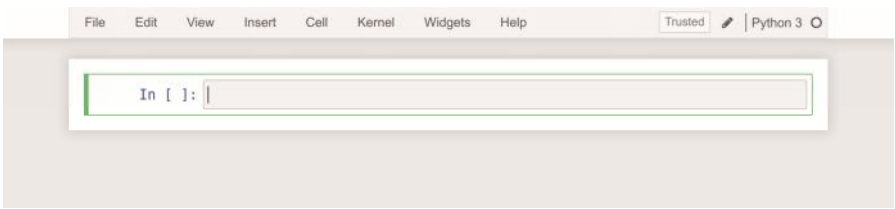


图 2.4　Jupyter Notebook 界面

Jupyter Notebook 界面由以下部分组成。

- Jupyter Notebook 的名称。默认为 Untitiled，也可通过单击重命名。
- 主工具栏。提供保存、导出、重载 Jupyter Notebook 以及重启内核等选项。
- 能够实现代码单元格的复制、粘贴、剪切、上下移动等。
- Jupyter Notebook 的主要区域。包含了内容编辑区。

工具栏的功能较为常规，结合图标就可快速探索 Jupyter Notebook 的功能，这里不再赘述。要详细了解 Jupyter Notebook 或一些库的具体内容，可以使用菜单栏右侧的 Help 菜单。

Jupyter Notebook 的内容编辑区由一个个称为单元格的部分组成，每个单元格可以有不同的用途，最常用的两种单元格是代码单元格和 Markdown 单元格。

在代码单元格中可以输入任意代码并执行。例如，输入“1+2”并按组合键 Shift + Enter，就会执行单元格中的代码，并将光标移到一个新单元格中，结果如图 2.5 所示。

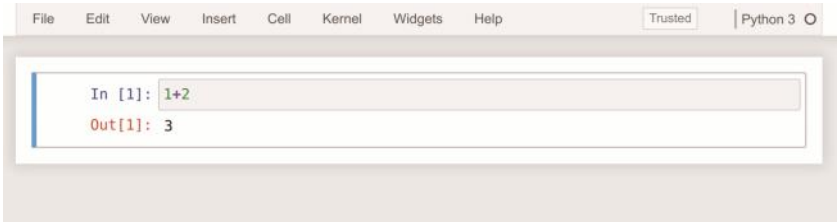


图 2.5　输入代码，并按组合键 Shift + Enter 执行计算，新生成一个单元格；按组合键 Ctrl + Enter 执行计算，光标仍然停留在当前单元格中

根据边框线的颜色，可以轻松地识别当前工作的单元格。接着，在第二个单元格中输入其他代码，如代码 2.12 所示。

代码 2.12 在单元格中输入代码

```
01: for i in range(3):
02:     print(i,end=" ")
```

对上面的代码求值，得到图 2.6 所示的结果。

图 2.6 代码求值结果

另外，Jupyter Notebook 有一个非常有趣的特性，即可以修改之前的单元格，对其重新进行计算，而不需要重新运行 Jupyter Notebook 内的所有代码。试着将光标移回第一个单元格，并将“1+2”修改为“2+3”，然后按组合键 Shift + Enter 重新计算该单元格，结果马上就更新成“5”。如果不想重新运行整个脚本，而只想用不同的参数测试某个程序，那么这个特性尤为方便。然而，也可重新计算整个 Jupyter Notebook，方法是选择菜单项 Cell → Run all。

前面介绍了如何输入代码。Jupyter Notebook 的强大之处是不仅能运行代码，而且可为代码加一些笔记或注释。为此，需要使用其他类型的单元格，即 Markdown 单元格。

选中第一个单元格，单击下拉框，然后选择 Markdown，该单元格就变成一个 Markdown 单元格，如图 2.7 所示。该单元格的语法遵循 Markdown 语法，支持输入公式、加入链接、将文本样式设为粗体或斜体、设置代码格式等。像代码单元格一样，按组合键 Shift + Enter 或 Ctrl + Enter 可以运行 Markdown 单元格，将 Markdown 显示为格式化文本。

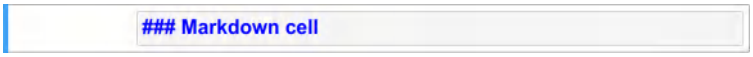


图 2.7 Markdown 单元格

Markdown 的详细介绍请参阅其他资料①，这里不做过多的介绍。

2.3.2 Jupyter Notebook 的快捷键

Jupyter Notebook 自带一组快捷键，能快速使用键盘与单元格交互，而无须使用鼠标和工具栏。熟悉这些快捷键需要花一些时间，但是熟练掌握后，将大大加快在 Jupyter Notebook 中编写代码和文档的速度。这里不详细介绍所有的快捷键，读者可在单元格呈蓝色（切换方式是单击单元格外的任何位置）时按“h”键查看，详细快捷键如图 2.8 所示。

① Markdown 中文网http://markdown.p2hp.com/。

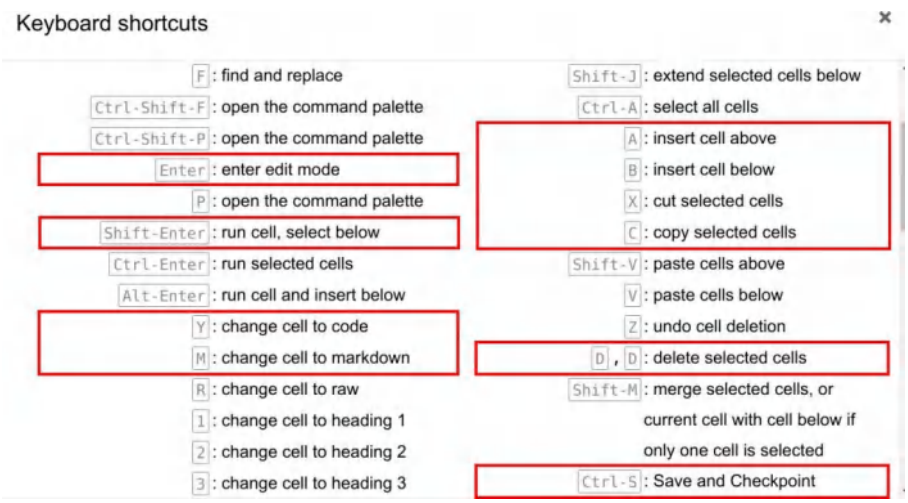


图 2.8　Jupyter Notebook 的快捷键，红框中的是常用快捷键

2.3.3　Magic 关键字

Magic 关键字是 Jupyter Notebook 的一些高级用法，可以运行特殊的命令，或者控制 Jupyter Notebook。例如，在 Jupyter Notebook 中可用`%matplotlib` 将 matplotlib 设置为以交互方式工作。

Magic 命令的前面有一个或两个百分号（`%`或`%%`），分别代表行 Magic 命令和单元格 Magic 命令。行 Magic 命令仅用于编写 Magic 命令时所在的行，单元格 Magic 命令则用于整个单元格。

例如，要测算整个单元格的运行时间，可以使用`%%time`，如图 2.9 所示。

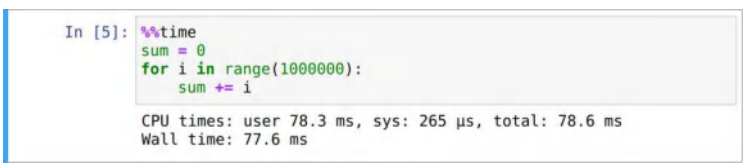


图 2.9　Jupyter Notebook 中的记时功能

当然，还有很多 Magic 关键字可供使用，使用`%lsmagic` 可以查看所有的 Magic 命令。

2.4　Python 基础

开始 Python 的第一步是打开 Python 的交互界面，可以选择 python 或 ipython。打开 Python 后，开始编写第一个程序“Hello World!”，在 Python 交互界面输入代码，如代码 2.13 所示。

代码 2.13　第一个程序

```
01: print("Hello World!")
```

输出为

```
Hello World!
```

Python 的强大功能不仅体现为 Python 语言本身，而且体现为拥有大量优质的第三方库。下面演示如何加载第三方库，如代码 2.14 所示。

代码 2.14　加载第三方库

```
01: import this
```

上面用 import 指令加载一个第三方库后，窗口输出由 Tim Peters 编写的“Python 之禅”，它详细介绍了 Python 的编程哲学，如图 2.10 所示。

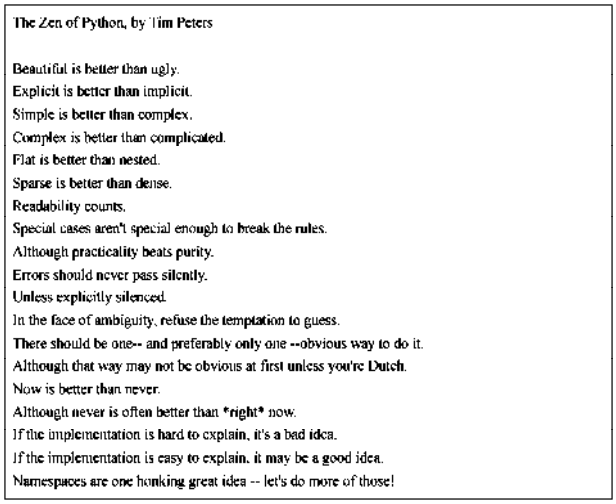


图 2.10　英文版“Python 之禅”

图 2.10 翻译为中文后，如图 2.11 所示。

优美胜于丑陋（Python 以编写优美的代码为目标）。
明了胜于晦涩（优美的代码应当是明了的，命名规范，网格相似）。
简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）。
复杂胜于凌乱（如果复杂不可避免，那么代码间也不能有难懂的关系，要保持接口简洁）。
扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）。
间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）。
可读性很重要（优美的代码是可读的）。
即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）。
不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except: pass 网格的代码）。
当存在多种可能时，不要尝试去猜测，而要尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）。
虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido）。
做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）。
如果你无法向人描述你的方案，那么肯定不是一个好方案，反之亦然（方案测评标准）。
命名空间是种绝妙的理论，我们应当多加利用（倡导与号召）。

图 2.11　中文版“Python 之禅”

2.4.1　变量

用于表示某个对象或值的名称称为变量。在 Python 中，可用如下方式声明变量并为其赋值，如代码 2.15 所示。

代码 2.15　变量声明与赋值

```
01: x = 2
02: y = 5
03: xy = 'Hey'
04: print(x+y, xy)
```

输出为

```
7 Hey
```

注意：Python 中的变量赋值不需要类型声明，因为 Python 是弱类型语言，同类语言还有 JavaScript、PHP 等。弱类型语言的特点如下：①变量无须声明就可直接赋值，对于一个不存在的变量赋值相当于定义一个新变量；②变量类型可随时改变，一个变量可以赋值为整数，然后赋值为字符串等。

2.4.2　运算符

1. 算术运算符

算术运算符即数学运算符，用来对数字进行数学运算，如加、减、乘、除。Python 的算术运算符和其他语言的类似，表 2.1 中小结了 Python 中的基本运算符。

表 2.1　Python 中的基本运算符

符　号	运　算
+	加法
-	减法
/	除法
%	取余
*	乘法
//	整数除法
**	幂

代码 2.16 演示了基本的运算。

代码 2.16　基本的运算

```
01: 1+2
02: 2-1
03: 1*2
04: 1/2
```

输出为

```
3
1
2
0.5
```

注意：Python 3 之后的版本自动地将整数除法转换为浮点数除法，因此 1/2 的结果是 0.5，以与人类的直觉一致。不过，为了程序的可读性，做除法运算时最好还是显式地写为浮点数，如 1.0/2。

除法和取余运算符示例如代码 2.17 所示。

代码 2.17　除法和取余运算符示例

```
01: 1.0/2
02: 1/2.0
03: 15%10
```

输出为

```
0.5
0.5
5
```

地板除法（Floor Divide）是指舍去结果的小数部分的除法，如代码 2.18 所示。

代码 2.18　地板除法

```
01: 2.8//2.0
```

输出为

```
1.0
```

2. 关系运算符

关系运算符也称比较运算符，用于对常量、变量或表达式的结果进行大小比较。当这种比较成立时，返回 True（真），否则返回 False（假）。表 2.2 中小结了 Python 的关系运算符。

表 2.2 Python 的关系运算符

符　号	运　算
==	是否相等，相等返回 True
!=	是否不等，不等返回 True
<	是否小于
>	是否大于
<=	小于或等于
>=	大于或等于

赋值和关系运算符示例如代码 2.19 所示。

代码 2.19 赋值和关系运算符示例

```
01: z=1
02: z==1
```

输出为

```
True
```

下面使用大于运算符进行测试，如代码 2.20 所示。

代码 2.20 大于运算符

```
01: z > 1
```

输出为

```
False
```

2.4.3 内置函数

函数能提高应用的模块性和代码的重复利用率。Python 提供了许多内置函数，下面介绍几个常用的内置函数。

1. range()函数

range()函数输出指定范围内的整数，是 Python 中的常用函数之一。它还可通过指定特定范围内的两个数字之差来生成一个序列，元素以迭代容器的形式返回，如代码 2.21 所示[①]。

代码 2.21 range()函数

```
01: print(list(range(3)))
02: print(list(range(2,9)))
03: print(list(range(2,27,8)))
```

输出为

```
[0, 1, 2]
[2, 3, 4, 5, 6, 7, 8]
[2, 10, 18, 26]
```

2. int()函数

int()函数将字符串或浮点数转换为整数。它有两个参数输入，一个是不同数字系统中的值，另一个是它的基数，如代码 2.22 所示。

代码 2.22 int()函数

```
01: print(int('010',8))
02: print(int('0xaa',16))
03: print(int('1010',2))
```

[①] 为了可视化迭代容器，此处使用 list()将迭代容器转换成列表进行打印。

输出为

```
8
170
10
```

类似的函数还有用于转换为二进制数的 bin()、用于转换为浮点数的 float()、返回值是当前整数对应的 ASCII 字符的 chr()、用于获得字符的 ASCII 值的 ord()。

3. round()函数

round()函数将输入值四舍五入为指定的位数和最接近的整数，如代码 2.23 所示。

代码 2.23　round()函数

```
01: print(round(5.6231))
02: print(round(4.55892, 2)) # 四舍五入到小数点后的第二位
```

输出为

```
6
4.56
```

4. type()函数与 isinstance()函数

由于 Python 是脚本语言，变量的类型不需要事先定义，所以有时候需要判断变量的类型，这就要用到类型判断函数 type()，该函数返回给定变量的类型。另外一个判断类型的函数是 isinstance()，它判断给定的变量是否是给定的类型，若是，则返回 True；这个函数还可同时检查多个类型。如代码 2.24 所示。

代码 2.24　type()函数和 isinstance()函数

```
01: print(type(1))
02: print(isinstance(1, int))
03: print(isinstance(1.0,int))
04: print(isinstance(1.0,(int,float)))
```

输出为

```
<class 'int'>
True
False
True
```

Python 的常用类型如表 2.3 所示。

表 2.3　Python 的常用类型

类型名字	解　　释
int	整数类型
float	浮点数类型
str	字符串类型
list	列表类型
tuple	元组类型
dict	字典类型
set	集合类型

2.5　print()函数

print()是 Python 的内置函数，主要用于打印变量的值，是 Python 编程时常用的函数，基本的用法如代码 2.25 所示。

代码 2.25　print()函数

```
01: print("Hello  World")
02: print("Hello",  "World")
03: print("Hello" + "World")
04: print("Hello %s" % "World")
```

输出为

```
Hello World
```

```
HelloWorld
HelloWorld
Hello World
```

在 Python 中，单引号（'）、双引号（"）和三引号（'''或"""）用于表示字符串。大部分单引号用于声明一个字符。声明一行时使用双引号；声明段落/多行字符串时使用三引号，如代码 2.26 所示。

代码 2.26 print()函数输出多行字符串

```
01: print("""My name is Jack
02:
03: I love Python.""")
```

输出为

```
My name is Jack
I love Python.
```

打印字符串中的“%s”用于引用包含字符串的变量，如代码 2.27 所示。

代码 2.27 %s 的用法

```
01: string1 = "World"
02: print("Hello  %s"  %  string1)
```

输出为

```
Hello World
```

这里的用法和 C 语言中是一样的。字符串中的控制符如表 2.4 所示，一些示例如代码 2.28 所示。

表 2.4 字符串中的控制符

符 号	运 算
%s	字符串
%d	整数
%f	浮点数
%o	八进制数
%x	十六进制数
%e	科学计数

代码 2.28 print()函数示例

```
01: print("Actual Number = %d" % 18)
02: print("Float of the number = %f" % 18)
03: print("Octal equivalent of the number = %o" % 18)
04: print("Hexadecimal equivalent of the number = %x" % 18)
05: print("Exponential equivalent of the number = %e" % 18)
```

输出为

```
Actual Number = 18
Float of the number = 18.000000
Octal equivalent of the number = 22
Hexadecimal equivalent of the number = 12
Exponential equivalent of the number = 1.800000e+01
```

引用多个变量时，要使用圆括号括起多个变量，将多个变量转换为一个元组，如代码 2.29 所示。

代码 2.29 引用多个变量

```
01: print("Hello %s %s" % ("World", "!"))
```

输出为

```
Hello World !
```

2.6 数据结构

数据结构是计算机存储、组织数据的方式，即相互之间存在一种或多种特定关系的数据元素的

集合。Python 在其标准库中提供了大量的数据结构，使用内置的几种数据结构可以方便、快捷地完成复杂程序的编写。

2.6.1　列表

列表是最常用的数据结构之一，可视为用方括号括起来的数据序列，数据之间用逗号分隔，并且这些数据都可通过其索引值来访问。

声明列表时，只需将变量等同于[]或 list，如代码 2.30 所示。

代码 2.30　声明列表

```
01: a = []
02: print(type(a))
```

输出为

```
<class 'list'>
```

可以直接将数据序列分配给列表 x，如代码 2.31 所示。

代码 2.31　列表初始化

```
01: x = ['apple', 'orange', 'peach']
02: print(x)
```

输出为

```
['apple', 'orange', 'peach']
```

1. 索引

在 Python 中，索引从 0 开始编号。在前面包含三个元素的列表 x 中，apple 的索引值为 0，orange 的索引值为 1。访问索引的用法如代码 2.32 所示。

代码 2.32　访问索引

```
01: print(x[0])
```

输出为

```
'apple'
```

索引也可反序访问，最后一个被访问的元素的索引值从 –1 开始。因此，索引值 –1 对应 peach，索引值 –2 对应 apple，如代码 2.33 所示。

代码 2.33　反序访问索引

```
01: x[-1]
```

输出为

```
'peach'
```

对于上例，有 x[0]=x[-2]和 x[1]=x[-1]，这个概念可以扩展到包含更多元素的列表。

下面再定义一个列表，如代码 2.34 所示。

代码 2.34　定义列表

```
01: y = ['carrot', 'potato']
```

这里声明了两个列表 x 和 y，每个列表都含有自己的数据。现在，这两个列表可再次放入另一个列表 z 中，列表 z 称为嵌套列表。这是与很多其他计算机语言不同的地方，即不要求列表的元素是相同类型的，因此编程时非常方便，而这也是 Python 对人友好的原因。下面给出一个示例，如代码 2.35 所示。

代码 2.35 生成列表索引

```
z = [x, y, 'Test']
print(z)
print(z[0][1])
```

输出为

```
[['apple', 'orange', 'peach'], ['carrot', 'potato'], 'Test']
'orange'
```

在 Python 中，还可通过并排书写索引值来访问'orange'，如代码 2.36 所示。

代码 2.36 列表索引访问

```
z[0][0]
'apple'
```

如果列表中有另一个列表，就可通过执行 z[][]来访问最里面的值。

2. 切片

索引只限于访问单个元素，而切片则访问列表内的一系列数据，即切片返回一个子列表。

切片是通过定义切片列表内需要的父列表中的第一个元素和最后一个元素的索引值来完成的，写为“父列表[a:b]”，其中 a 和 b 是父列表的索引值。当索引值 a 未定义时，就默认为列表中的第一个值；当 b 未定义时，就默认为列表中的最后一个值。切片访问如代码 2.37 所示。

代码 2.37 切片访问

```
num = [0,1,2,3,4,5,6,7,8,9]
print(num[1:4])
print(num[0:4])
print(num[4:])
print(num[0:])
print(num[:])
print(num)
```

输出为

```
[1, 2, 3]
[0, 1, 2, 3]
[4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

还可指定相隔的固定长度（即步长）对父列表进行切片，如代码 2.38 所示。

代码 2.38 列表切片

```
num[:9:3]
```

输出为

```
[0, 3, 6]
```

3. 列表的内置函数

1）len()函数

要求出列表的长度或者列表中元素的数量，可以使用 len()函数，如代码 2.39 所示。

代码 2.39　len()函数

```
01: len(num)
```

输出为

```
10
```

2）min()函数和 max()函数

若列表元素均为整数，则 min()和 max()分别返回列表中的最大值和最小值，如代码 2.40 所示。

代码 2.4　min()函数和 max()函数

```
01: num = [0,1,2,3,4,5,6,7,8,9]
02: min(num)
03: max(num)
```

输出为

```
0
9
```

在以字符串为元素的列表中，也可使用 max()和 min()函数，max()返回 ASCII 码值最大的元素，min()返回 ASCII 码值最小的元素。

注意：每次只考虑每个元素的第一个索引，当它们的值相同时才考虑第二个索引，以此类推。

对于字符串列表，min()函数和 max()函数的用法示例如代码 2.41 所示。

代码 2.4　字符串列表的 min()和 max()函数

```
01: mlist = ['bzaa','ds','nc','az','z','klm']
02: print(max(mlist))
03: print(min(mlist))
```

输出为

```
z
az
```

这里考虑的是每个元素的第一个索引，z 有最大的 ASCII 码值，因此被返回，最小的 ASCII 码值是 az。然而，如果数字被声明为字符串呢？下面给出示例，如代码 2.42 所示。

代码 2.42　字符串列表的 min()和 max()函数

```
01: nlist = ['1','94','93','1000']
02: print(max(nlist))
03: print(min(nlist))
```

输出为

```
94
1
```

即使数字是在字符串中声明的，也考虑每个元素的第一个索引，且相应地返回最大值和最小值。

前面是根据列表中元素的值进行判断的，要找到字符串长度最大的字符串元素，就要在 max()和 min()函数中声明参数'key=len'，示例如代码 2.43 所示。

代码 2.43 声明参数

```
01: names = ['Earth','Jet','Air','Fire','Water']
02: print(max(names, key=len))
03: print(min(names, key=len))
```

输出为

```
Earth
Jet
```

注意，即使 Water 与 Earth 的长度相同，max()函数也只返回 Earth，因为当两个或多个元素的长度相同时，max()函数和 min()函数返回第一个元素。

注意：也可使用任何其他内建函数或后面介绍的 lambda 函数来代替 len。

3）列表连接

列表可以使用“+”连接起来，连接后的列表包含所添加列表中的所有元素，如代码 2.44 所示。

代码 2.44 列表连接

```
01: print([1,2,3] + [5,4,7])
```

输出为

```
[1, 2, 3, 5, 4, 7]
```

4）in

在 Python 中编程时，常要检查列表中是否存在特定的元素，如代码 2.45 所示。

代码 2.45 判断元素是否在列表中

```
01: names = ['Earth','Air','Fire','Water']
```

上述代码检查 Fire 和 Rajath 是否出现在列表中。传统方法是用 for 循环遍历列表并用 if 语句进行判断，但在 Python 中可以使用“a in b”的方法，即如果 a 在 b 中出现，则返回 True，否则返回 False，如代码 2.46 所示。

代码 2.46 判断元素是否在列表中

```
01: 'Fir' in names
02: 'Fire' in names
03: 'fire' in names
04: 'Rajath' in names
```

输出为

```
False
True
False
False
```

5）list()函数

使用 list()函数可将字符串转换为列表，如代码 2.47 所示。

代码 2.47 类型转换

```
01: list('hello')
```

输出为

```
['h', 'e', 'l', 'l', 'o']
```

6）append()函数

append()函数在列表的最后添加一个元素，如代码 2.48 所示。

代码 2.48　在列表中添加元素

```
01: lst = [1,1,4,8,7]
02: lst.append(1)
03: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, 1]
```

append()函数还可用于在末尾添加列表，观察发现得到的列表是嵌套列表，如代码 2.49 所示。

代码 2.49　在列表中添加列表

```
01: lst1 = [5,4,2,8]
02: lst.append(lst1)
03: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, 1, [5, 4, 2, 8]]
```

7）count()函数

count()函数用于计算列表中特定元素的数量，如代码 2.50 所示。

代码 2.50　计算列表中特定元素的数量

```
01: lst.count(1)
```

输出为

```
3
```

8）extend()函数

如果有第二个列表，希望将它加入第一个列表，但不希望第二个列表作为整体嵌入第一个列表，而只将第二个列表中的元素放入第一个列表的末尾，则可以使用 extend()函数，如代码 2.51 所示。

代码 2.51　列表扩展

```
01: # 将 lst1 中的元素扩展到 lst 中
02: lst.extend(lst1)
03: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, [5, 4, 2, 8], 5, 4, 2, 8]
```

9）insert(x,y)函数

insert(x,y)函数在指定的索引值 x 处插入元素 y。相反，append()函数只将元素插入列表的最后，如代码 2.52 所示。

代码 2.52　插入元素

```
01: lst.insert(5, 'name')
02: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, 'name', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

insert(x,y)函数插入但不替换元素。要替换某个元素，只需将值赋给特定的索引，如代码 2.53 所示。

代码 2.53 列表中元素的赋值

```
01: lst[5] = 'Python'
02: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

10）pop()函数

pop()函数返回列表中的最后一个元素，并在列表中“删除”它，类似于堆栈操作。因此，列表可作为堆栈使用，如代码 2.54 所示。

代码 2.54 弹出元素

```
01: lst.pop()
02: print(lst)
```

输出为

```
[1, 1, 4, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 4, 2]
```

该函数也可通过给定索引值来弹出与该索引值对应的元素，如代码 2.55 所示。

代码 2.55 给定索引值弹出元素

```
01: print(lst.pop(2))
02: print(lst)
03: print(lst)
04: print(lst.pop(-2))
```

输出为

```
4
[1, 1, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 4, 2]
4
[1, 1, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 2]
```

11）remove()函数

除了 pop()函数根据索引值删除对应的元素，还可使用 remove()函数指定元素本身来删除元素，如代码 2.56 所示。

代码 2.56 列表删除元素

```
01: lst.remove('Python')
02: print(lst)
```

输出为

```
[1, 1, 8, 7, 1, [5, 4, 2, 8], 5, 2]
```

12）del

del 使用索引值删除特定的元素，可以代替 remove()函数，如代码 2.57 所示。

代码 2.57　使用 del 删除元素

```
01: print(lst)
02: del lst[5]
03: print(lst)
```

输出为

```
[1, 1, 8, 7, 1, [5, 4, 2, 8], 5, 2]
[1, 1, 8, 7, 1, 5, 2]
```

13）sort()函数

Python 提供内置函数 sort()按升序排列元素，如代码 2.58 所示。

代码 2.58　列表排序

```
01: lst = [1, 4, 8, 8, 10]
02: lst.sort()
03: print(lst)
```

输出为

```
[1, 4, 8, 8, 10]
```

sort()函数有一个默认的参数 reverse = False，表示按升序排序。要降序排序，可将这个默认参数设置为 reverse = True。例如，对于包含字符串元素的列表，sort()根据它们的 ASCII 码值以升序方式排序，而通过设置 reverse = True 以降序方式排列。

如果要根据长度排序，就应该如代码 2.59 所示的那样设置参数 key=len。

代码 2.59　根据长度排序

```
01: names.sort(key=len)
02: print(names)
03: names.sort(key=len,reverse=True)
04: print(names)
```

输出为

```
['peach', 'apple', 'orange']
['orange', 'peach', 'apple']
```

4. 复制列表

大多数人开始学习 Python 时都会犯一个错误——难以区分对象的赋值和复制。考虑代码 2.60 所示的例子。

代码 2.60　列表复制

```
01: lista = [2,1,4,3]
02: listb = lista
03: print(listb)
```

输出为

```
[2, 1, 4, 3]
```

首先声明了一个列表 lista=[2,1,4,3]，并通过赋值将该列表复制到 listb。下面对 lista 执行一些操作，如代码 2.61 所示。

代码 2.61　列表操作

```
01: lista.pop()
```

```
02: print(lista)
03: lista.append(9)
04: print(lista)
05: print(listb)
```

输出为

```
[2, 1, 4]
[2, 1, 4, 9]
[2, 1, 4, 9]
```

虽然没有对 listb 执行任何操作，但它发生了变化，因为 lista、listb 指向相同的内存空间。如果要求 listb 是一个独立的列表，那么应该如何解决这个问题呢？

在切片中，“父列表[a:b]”从父列表中返回一个起始索引为 a、结束索引为 b 的列表，未提及 a 和 b 时，默认将第一个元素和最后一个元素赋给 lista 和 listb。因此，一种方法是通过切片生成一个新的列表并赋值给 listb，如代码 2.62 所示。

代码 2.62 复制列表

```
01: lista = [2,1,4,3]
02: listb = lista[:]
03: print(listb)
04: lista.pop()
05: print(lista)
06: lista.append(9)
07: print(lista)
08: print(listb)
```

输出为

```
[2, 1, 4, 3]
[2, 1, 4]
[2, 1, 4, 9]
[2, 1, 4, 3]
```

注意：为了使通用性更好，可以使用 Python 库 copy 中的 deepcopy()函数。

2.6.2 元组

元组与列表相似，唯一的区别是列表中的元素可以更改，而元组中的元素不能更改。为了更好地理解这一点，回顾 divmod()函数，如代码 2.63 所示。

代码 2.63 元组示例

```
01: xyz = divmod(10,3)
02: print(xyz)
03: print(type(xyz))
04: xyz[0]=10
```

输出为

```
(3, 1)
<class 'tuple'>
TypeError: 'tuple' object does not support item assignment
```

10 除以 3 的商是 3，余数是 1，作为结果，这些值不能改变。因此，divmod()函数以元组形式返回这些值，而最后一步报错，因为元组中的元素不能更改。

要定义元组，可将一个变量写入()或 tuple()，如代码 2.64 所示。

代码 2.64　定义元组

```
01: tup = ()
02: tup2 = tuple()
```

要直接声明元组，可在数据末尾使用逗号，如代码 2.65 所示。

代码 2.65　直接声明元组

```
01: 27,
```

输出为

```
(27,)
```

声明元组时，可以赋值，它接受列表作为输入并将其转换为元组，或者接受字符串并将其转换为元组，如代码 2.66 所示。

代码 2.66　转换成元组

```
01: tup3 = tuple([1,2,3])
02: print(tup3)
03: tup4 = tuple('Hello')
04: print(tup4)
```

输出为

```
(1, 2, 3)
('H', 'e', 'l', 'l', 'o')
```

它遵循与列表相同的索引和切片，如代码 2.67 所示。

代码 2.67　元组切片

```
01: print(tup3[1]) 2: tup5 = tup4[:3] 3: print(tup5)
```

输出为

```
2
('H', 'e', 'l')
```

1. 元组间的映射

通过元组间的赋值，可以实现多个元素同时赋值或调换顺序。下例用一条语句直接给变量 a,b,c 赋值，如代码 2.68 所示。

代码 2.68　多元素赋值

```
01: (a,b,c) = ('alpha','beta','gamma')
02: print(a,b,c)
03: d = tuple('RajathKumarMP')
04: print(d)
```

输出为

```
alpha beta gamma
('R', 'a', 'j', 'a', 't', 'h', 'K', 'u', 'm', 'a', 'r', 'M', 'P')
```

2. 元组内置函数

1）count()函数

count()函数计算元组中存在的指定元素的数量，如代码 2.69 所示。

代码 2.69 元组的元素计数函数

```
01: d.count('a')
```

输出为

```
3
```

2）index()函数

index()函数返回指定元素的索引。找到指定的元素时，返回指定元素的第一个元素的索引，未找到指定的元素时则报错，如代码 2.70 所示。

代码 2.70 元组的元素索引函数

```
01: d.index('a')
```

输出为

```
1
```

2.6.3 集合

集合函数 set()主要用于消除序列/列表中的重复元素，还可执行一些标准的集合操作。

1. 基础操作

集合被声明为 set()时，将初始化一个空集。也可执行 set([sequence])来声明一个包含给定元素的集合，如代码 2.71 所示。

代码 2.71 集合的定义

```
01: set1 = set()
02: print(type(set1))
```

输出为

```
<class 'set'>
```

代码 2.72 演示了生成包含给定列表中元素的集合。

代码 2.72 生成集合

```
01: set0 = set([1,2,2,3,3,4])
02: print(set0)
```

输出为

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

也可将元组转换为集合，如代码 2.73 所示。

代码 2.73 元组转换为集合

```
01: set1 = set((1,2,2,3,3,4))
02: print(set1)
```

输出为

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

重复两次的元素 2,3 只出现一次，因此一个集合中的每个元素都是不同的。

2. 内置函数

先声明两个集合作为示例数据，如代码 2.74 所示。

代码 2.74　示例集合

```
01: set1 = set([1,2,3])
02: set2 = set([2,3,4,5])
```

1）union()函数

union()函数返回一个集合，该集合是两个集合的并集，如代码 2.75 所示。

代码 2.75　集合的并集

```
01: set1.union(set2)
```

输出为

```
{1, 2, 3, 4, 5}
```

2）add()函数

add()函数向集合中添加一个特定的元素，如代码 2.76 所示。

代码 2.76　集合中添加元素

```
01: print(set1)
02: set1.add(0)
03: print(set1)
```

输出为

```
{1, 2, 3}
{0, 1, 2, 3}
```

3）intersection()函数

intersection()函数输出一个集合，该集合是两个集合的交集，如代码 2.77 所示。

代码 2.77　集合的交集

```
01: set1.intersection(set2)
```

输出为

```
{2, 3}
```

4）difference()函数

difference()函数输出一个集合，其中包含在 set1 中而不在 set2 中的元素，如代码 2.78 所示。

代码 2.78　集合的差

```
01: print(set1)
02: print(set2)
03: set1.difference(set2)
```

输出为

```
{0, 1, 2, 3}
{2, 3, 4, 5}
{0, 1}
```

5）pop()函数

pop()函数用于删除集合中最小的元素，如代码 2.79 所示。

代码 2.79 集合弹出元素

```
01: set1=set([10, 9, 1, 2, 4])
02: set1.pop()
03: print(set1)
```

输出为

```
{2, 4, 9, 10}
```

6）remove()函数

remove()函数从集合中删除指定的元素，如代码 2.80 所示。

代码 2.80 集合删除元素

```
01: set1.remove(2)
02: print(set1)
```

输出为

```
{1, 4, 9, 10}
```

7）clear()函数

clear()函数用于清除集合中的所有元素并将集合设为空集，如代码 2.81 所示。

代码 2.81 集合清空

```
01: set1.clear()
02: print(set1)
```

输出为

```
set()
```

2.6.4 字符串

字符串（String）是字符序列，或者说是一串字符。字符只是一个符号，如英语有 26 个字符。Python 不支持单字符类型，单字符在 Python 中也作为一个字符串使用。将字符括在单引号或双引号中创建字符串。Python 中可以使用三引号，常用于表示多行字符串和文档字符串。

1. 基本用法

字符串的基本用法如代码 2.82 所示。

代码 2.82 字符串的基本用法

```
01: String0 = 'Python is beautiful'
02: String1 = "Python is beautiful"
03: String2 = '''Python
04: is
05: beautiful'''
```

使用 print()函数输出字符串，如代码 2.83 所示。

代码 2.83 字符串输出

```
01: print(String0, type(String0))
02: print(String1, type(String1))
```

```
03: print(String2, type(String2))
```

输出为

```
Python is beautiful <class 'str'>
Python is beautiful <class 'str'>
Python is
beautiful <class 'str'>
```

字符串索引和切片类似于前面详细解释的列表，如代码 2.84 所示。

代码 2.84　字符串切片访问

```
01: print(String0[4])
02: print(String0[4:])
```

输出为

```
o
on is beautiful
```

2. 内置函数

下面介绍关于字符串的一些常用函数。注意，这里所用的字符串变量仍承接上文的定义。

1）find()函数

find()函数的具体用法如代码 2.85 所示。

代码 2.85　查找子字符串 1

```
01: str.find(str, beg = 0, end = len(string))
```

第二个和第三个参数的默认值分别为 0 和被搜索字符串的长度。返回值是在字符串中找到的给定数据的索引值，如果未找到，那么返回-1。

注意：不要混淆 find()函数返回的“-1”与列表索引值“-1”，此处返回的“-1”表示没有找到符合要求的子字符串。

代码 2.86 给出了一个简单的例子。

代码 2.86　查找子字符串 2

```
01: print(String0)
02: print(String0.find('are'))
03: print(String0.find('is'))
```

输出为

```
Python is beautiful
-1
7
```

可以看出未找到时返回-1，找到时返回第一个元素的索引。代码 2.87 给出了另一个例子。

代码 2.87　查找子字符串 3

```
01: print(String0[7])
```

输出为

```
i
```

下面将后两个参数用上，如代码 2.88 所示。

代码 2.88 查找子字符串 4

```
01: print(String0.find('is',1))
02: print(String0.find('is',1,3))
```

输出为

```
7
-1
```

第一行代码在从 1 到字符串末尾的区间上查找，第二行代码则在索引区间 1～3 上查找。

2）index()函数

index()和 find()函数的工作方式相同，唯一的区别是当输入的元素未在字符串中找到时，find()函数返回-1，而 index()函数抛出 ValueError，如代码 2.89 所示。

代码 2.89 定位子字符串

```
01: print(String0.index('Python'))
02: print(String0.index('is',0))
03: print(String0.index('Python',10,20))
```

输出为

```
0
7
ValueError: substring not found
```

可以看出第三行代码报错，因为未找到子字符串。

3）split()函数

split()函数以指定字符串为依据分割字符串对象，并将分割后的字符串序列以列表的形式返回。我们可将它视为 join()的反函数，如代码 2.90 所示。

代码 2.90 字符串分割

```
01: c = "Python is beautiful"
02: d = c.split(' ')
03: print(d)
```

输出为

```
['Python', 'is', 'beautiful']
```

在 split()函数中，还可指定分割字符串的次数，或者新返回列表中应包含的元素数量。元素的数量总比指定的数量多 1，因为它被分割了指定的次数，如代码 2.91 所示。

代码 2.91 字符串分割

```
01: e = c.split(' ', 1)
02: print(e)
03: print(len(e))
```

输出为

```
['Python', 'is beautiful']
2
```

4）join()函数

join()函数在输入字符串的列表之间添加给定的字符串，如代码 2.92 所示。

代码 2.92　字符串列表合并 1

```
01: 'a'.join('*_-')
02: '\n'.join(['1', '2'])
```

输出为

```
'*a_a-'
'1\n2'
```

这里，'*_-'是输入字符串，字符'a'被添加到每个元素之间。join()函数也可用来将列表转换为字符串，如代码 2.93 所示。

代码 2.93　字符串列表合并 2

```
01: a = list(String0)
02: print(a)
03: b = ''.join(a)
04: print(b)
```

输出为

```
['P', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 'b', 'e', 'a', 'u', ' t', 'i', 'f', 'u', 'l']
Python is beautiful
```

5）lower()函数

lower()函数的作用是将大写字母转换为小写字母，如代码 2.94 所示。

代码 2.94　字母转换为小写

```
01: print(String0)
02: print(String0.lower())
```

输出为

```
Python is beautiful
python is beautiful
```

6）upper()函数

upper()函数的作用与 lower()函数的相反，即将小写字母转换为大写字母，如代码 2.95 所示。

代码 2.95　字母转换为大写

```
01: String0.upper()
```

输出为

```
'PYTHON IS BEAUTIFUL'
```

7）replace()函数

replace()函数的作用是将一个字符串替换为另一个字符串，如代码 2.96 所示。

代码 2.96　字符串替换

```
01: print(String0.replace('Python', 'PYTHON'))
```

输出为

```
'PYTHON is beautiful'
```

8）strip()函数

strip()函数用于从右端和左端删除不需要的元素，如果未指定字符，则默认删除数据左边和右

边的所有空格、制表符、换行符，如代码 2.97 所示。

代码 2.97 字符串清理

```
01: f = ' hello     '
02: print(f.strip())
```

输出为

```
'hello'
```

2.6.5 字典

字典（Dictionary）是 Python 提供的一种常用数据结构，由键（key）和值（value）组成，键和值中间用冒号“:”分隔，数据项之间用逗号分隔，整个字典被花括号“{ }”括起。由于可以使用定义的字符串索引特定的数据，所以字典更像数据库。Python 的字典使用键值对即 key-value 来存储，因此具有极快的查找速度。

1. 基础操作

要定义一个字典，可让一个变量等于{}或 dict()，如代码 2.98 所示。

代码 2.98 字典定义

```
01: d0 = {}
02: d1 = dict()
03: print(type(d0), type(d1))
```

输出为

```
<class 'dict'> <class 'dict'>
```

1）字典构建

字典的工作方式类似于列表，但增加了自己分配索引样式的功能。自己分配的索引就是“键”，而后面等于的元素就是“值”，通过对键进行搜索可以很快地匹配到它的值，如代码 2.99 所示。

代码 2.99 字典创建

```
01: d0['One'] = 1
02: d0['OneTwo'] = 12
03: print(d0)
04: d1 = {"key1":1, "key2":[1,2,4], 3:(1, 4, 6)}
05: print(d1)
```

输出为

```
{'One': 1, 'OneTwo': 12}
{'key1': 1, 'key2': [1, 2, 4], 3: (1, 4, 6)}
```

可以通过设为'One'的索引值来访问 1，如代码 2.100 所示。

代码 2.100 字典访问元素

```
01: print(d0['One'])
```

输出为

```
1
```

也可通过循环来构建字典，如代码 2.101 所示。

代码 2.101　字典生成

```
# 字典直接生成
a1 = {names[i]:numbers[i] for i in range(len(names))}
print(a1)

# 传统方法
for i in range(len(names)):
    a1[names[i]]  =  numbers[i]
print(a1)
```

输出为

```
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```

2）字典合并

两个相关的列表可以合并成一个字典，这个功能由 zip()函数实现，如代码 2.102 所示。

代码 2.102　字典合并

```
names = ['One', 'Two', 'Three', 'Four', 'Five']
numbers = [1, 2, 3, 4, 5]

d3 = {names[i]:numbers[i] for i in range(len(names))}
print(d3)

d2 = zip(names,numbers)
print(dict(d2))
```

输出为

```
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```

上述代码首先定义两个列表并一一比对输出，通过 zip()函数处理后，再由 dict()函数将其转换为字典并输出。

2. 内置函数

这里主要介绍字典中较为常用的内置函数。

1）clear()函数

clear()函数用于清除所创建的整个字典，如代码 2.103 所示。

代码 2.103　字典清空

```
a1 = {1:10, 2:20}
a1.clear()
print(a1)
```

输出为

```
{}
```

2）values()函数

values()函数返回一个包含字典中所有元素的值的列表，如代码 2.104 所示。

代码 2.104　所有元素的值的列表

```
01: a1.values()
```

输出为

```
dict_values([1, 2, 3, 4, 5])
```

3）keys()函数

keys()函数返回包含所有元素的索引或键的列表，如代码 2.105 所示。

代码 2.105 所有元素的索引或键的列表

```
01: a1.keys()
```

输出为

```
dict_keys(['One', 'Two', 'Three', 'Four', 'Five'])
```

4）items()函数

items()函数返回一个包含可遍历键值对的列表，与用 zip()函数的结果相同，见代码 2.106。

代码 2.106 字典中键值对的列表

```
01: for (k,v) in d3.items():
02:     print("[%6s] %d" % (k, v))
```

输出为

```
[   One] 1
[   Two] 2
[ Three] 3
[  Four] 4
[  Five] 5
```

5）pop()函数

pop()函数删除字典中的给定键及对应的值，返回值为被删除的值。键值必须给出，否则会产生类型错误，如代码 2.107 所示。

代码 2.107 字典弹出元素

```
01: a2 = d3.pop('One')
02: print(a2)
```

输出为

```
1
```

2.7 控制流语句

一般情况下，程序按照语句编写顺序依次执行，形成标准的面向过程的结构化形式。然而，由于程序具有很强的逻辑性，有时需要根据某些条件选择性地执行某些语句或者跳过某些语句。控制流语句用于控制程序流程的选择、循环、转向和返回等，以实现程序的各种结构。Python 的流程控制语句和其他语言的类似，如 C 语言。下面主要根据不同之处介绍 Python 的特点。

注意：输入代码时，要注意缩进表示的代码块的所属关系，建议使用 4 个空格来缩进一层。

2.7.1 判断语句

对于条件判断，下面先介绍最经典的 if 语句，如代码 2.108 所示。

代码 2.108　if 的基本用法

```
if some_condition:
    algorithm
```

下面看一个实例，如代码 2.109 所示。

代码 2.109　if 用法实例

```
x = 4
if x < 10:
    print("Hello")
```

输出为

```
Hello
```

可以看出，因为 x<10，所以跳入分支语句，输出 Hello。

注意：Python 中没有其他语言中的代码块开始标识符和结束标识符，如 C/C++中的{或}，而使用代码缩进表示代码的层级关系。

1. if-else

if-else 的作用是，如果不符合 if 判断语句的条件，就不进入 if 分支而跳入 else 分支，如代码 2.110 所示。

代码 2.110　if-else 的基本用法

```
if some_condition:
    code_block1
else:
    code_block2
```

用法示例如代码 2.111 所示。

代码 2.111　if-else 用法示例

```
x = 4
if x > 10:
    print("hello")
else:
    print("world")
```

输出为

```
world
```

可以看出，由于 x=4，不满足 x>10，所以跳入 else 分支输出 world。

2. if-elif

elif 是 else-if 的缩写，即在 if-else 的基础上多加了一个分支，以增强其灵活性，如代码 2.112 所示。

代码 2.112　if-elif 的基本用法

```
if some_condition:
    algorithm
elif some_condition:
```

```
    algorithm
else:
    algorithm
```

用法示例如代码 2.113 所示。

代码 2.113 if-elif 用法示例

```
x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
else:
    print("x=y")
```

输出为

```
x<y
```

if 语句可以嵌套，即在 if 语句中可以嵌入其他的 if 语句，如代码 2.114 所示。

代码 2.114 if 的嵌套

```
x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
    if x==10:
        print("x=10")
    else:
        print("invalid")
else:
   print("x=y")
```

输出为

```
x<y x=10
```

2.7.2 循环语句

循环语句可以在某个条件下循环执行某段程序，以便重复处理的相同任务。循环语句主要分为两个模块：for 和 while。

1. for 循环

for 循环是迭代循环，在 Python 中相当于一个通用的序列迭代器，可以遍历任何有序序列，如 str、list、tuple 等，也可以遍历任何可迭代对象，如 dict，如代码 2.115 所示。

代码 2.115 for 循环

```
for variable in something:
    algorithm
```

使用示例如代码 2.116 所示。

代码 2.116　for 循环示例

```
01: for i in range(5):
02:     print(i)
```

range(5)表示生成一个长度为 5 的迭代器，通过 for 循环依次遍历每个元素。输出为

```
0
1
2
3
4
```

也可以遍历一个给定的列表，如代码 2.117 所示。

代码 2.117　for 循环遍历列表

```
01: a = [1, 2, 5, 6]
02: for i in a:
03:     print(i)
```

输出为

```
1
2
5
6
```

由上面的例子可以发现，不同于 C 语言，Python 中的 for 循环偏向于自然语言，可以直接使用列表等作为对象进行遍历。

注意：range()函数的作用是返回一个可以迭代的对象，如 range(5)返回 0～4 的迭代器对象，range(1,5)返回 1～4 的迭代器对象。

for 循环也可以嵌套，如代码 2.118 所示。

代码 2.118　for 嵌套

```
01: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
02: # 第一重循环
03: for list1 in list_of_lists:
04:     # 第二重循环
05:     for x in list1:
06:         print(x)
07:     print()
```

输出为

```
1
2
3

4
5
6

7
```

```
8
9
```

2. while 循环

while 循环在某个条件下循环执行给定的代码块，如代码 2.119 所示。

代码 2.119 while 的用法

```
while some_condition:
    algorithm
```

使用示例如代码 2.120 所示。

代码 2.120 while 使用示例

```
i = 1
while i < 3:
    print(i ** 2)
    i = i+1
print('Bye')

# do-untile
while True:
    #do something
    i = i+1
    print('looping %3d' % i)

    # check stop condition
    if i >= 11: break
```

输出为

```
1
4
Bye
looping   4
looping   5
looping   6
looping   7
looping   8
looping   9
looping  10
looping  11
```

3. break 语句

break 语句在某个条件下退出循环，它只能用在循环中而不能单独使用；在嵌套循环中，break 语句只对最近的一层循环起作用，如代码 2.121 所示。

代码 2.121 break 的用法

```
for i in range(100):
    print(i)
    if i>=7:
        break
```

输出为

```
0
1
2
3
4
5
6
7
```

4. continue 语句

continue 语句与 C 语言中的使用方式相同，作用是跳出本次执行但不跳出整个循环，如代码 2.122 所示。

代码 2.122　continue 的用法

```
01: for i in range(10):
02:     if i>4:
03:         print("Although larger then 4, but continue.")
04:         continue
05:     elif  i<7:
06:         print(i)
```

输出为

```
0
1
2
3
4
Although larger then 4, but continue.
Although larger then 4, but continue.
Although larger then 4, but continue.
Although larger then 4, but continue.
Although larger then 4, but continue.
```

可以看出，当 i>4 时触发 continue，导致不再执行 print(i)。

5. 列表推导

Python 使用列表推导（List Comprehensions）模式，用一行代码就可生成所需的列表或字典等容器。例如，需要生成 2 的倍数时，可以先用 for 循环，如代码 2.123 所示。

代码 2.123　列表推导

```
01: res = []
02: for i in range(1,11):
03:     x = 2*i
04:     res.append(x)
05: print(res)
```

输出为

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

上面需要 4 行代码才能生成一个简单的列表，而使用 Python 可在定义变量时直接生成列表的内容，这种方法就是列表推导，即通过区间、元组、列表、字典和集合等数据类型快速生成一个满足指定要求的列表，如代码 2.124 所示。

代码 2.124 列表推导

```
01: [2*x for x in range(1,11) if x<5]
02:
03: print(a)
```

输出为

```
[2, 4, 6, 8]
```

列表推导的用法说明如图 2.12 所示，主要分为三部分：①元素的值：当前循环想要插入某个值，这个值可以是包含 x 的某个表达式，也可以是不包含 x 的某个表达式；②需要多少个元素：x 的取值范围是从 1 到 11，即需要循环 10 次；③判断本次循环是否插入新的元素：虽然共需要 10 次循环，但并非每次循环都必须插入一个新元素（列表不一定要包含 10 个数值），每次循环时需要按照某种判断条件，如当前循环的 x 是否小于 5，如果小于 5，就插入一个新值，否则不插入新值。

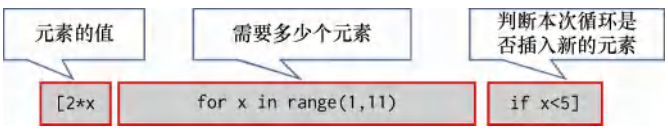


图 2.12 列表推导的用法说明

列表推导不仅适用于列表，而且适用于字典，如代码 2.125 所示。

代码 2.125 列表推导用于字典生成

```
01: a = {str(2*x):2*x for x in range(1,20) if x<=5}
02: print(a)
```

输出为

```
{'2': 2, '4': 4, '6': 6, '8': 8, '10': 10}
```

列表推导还可定义元组，如代码 2.126 所示。

代码 2.126 列表推导用于生成元组

```
01: tuple((2*x for x in range(1,20) if x<=5))
```

输出为

```
(2, 4, 6, 8, 10)
```

此外，列表推导还支持嵌套，如代码 2.127 所示。

代码 2.127 列表推导嵌套

```
01: [10*i+z for i in range(10) if i<5 for z in range(1,5)]
```

输出为

```
[1, 2, 3, 4, 11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43, 44]
```

可以看出，当 i<5 时，继续执行后面 z 的生成，列表继续增加；一旦不满足条件，循环就终止，列表推导随之结束。

2.8　函数

在一个程序中，有时需要重复执行一组语句，如果每次都写出这组语句，那么不仅乏味，而且编程效率很低，降低程序的可读性。为了抽取重复执行的语句，可以使用函数封装一组操作，并给出名称和参数列表作为函数的输入。Python 中的函数定义如代码 2.128 所示。

代码 2.128　函数定义

```
01: def func_name(arg1, arg2,..., argN):
02:     '''Document String'''
03:     statements
04:     return <value>
```

以上语法的理解如下：定义一个名为 func_name 的函数，它接受参数 arg1,arg2,...,argN，并在执行语句后返回一个值。下面通过一个具体的例子说明函数的用法和意义，如代码 2.129 所示。

代码 2.129　函数操作

```
01: print("Hey Python!")
02: print("Python, How do you do?")
```

输出为

```
Hey Python!
Python, How do you do?
```

如果不想每次都写出上面的两条语句，可以定义一个函数替代它，定义函数后，需要使用上方的两行代码时就只需写一行代码。下面定义一个函数 first_func()，如代码 2.130 所示。

代码 2.130　函数定义

```
01: def first_func():
02:     print("Hey Python!")
03:     print("Python, How do you do?")
04:
05: first_func()
06: funca = first_func
07: funca()
```

在上例中，除了直接调用函数名 first_func，将函数赋值给另一个变量 funca 同样可以调用和执行。输出为

```
Hey Python!
Python, How do you do?
Hey Python!
Python, How do you do?
```

2.8.1　函数的参数

first_func()每次都只打印固定的消息。一般来说，函数需要执行不同的内容，这时可让函数 first_func()接受一个存储名称的参数，然后打印相应的字符串。为此，需要在函数内添加一个参数，如代码 2.131 所示。

代码 2.131　函数参数

```
01: def first_func(username):
```

```
    print("Hey", username + "!")
    print(username + "," ,"How do you do?")
```

可以使用 input()函数输入用户的名称，如代码 2.132 所示。

代码 2.132 函数参数示例

```
name1 = input("Please enter your name : ")
```

输出为

```
Please enter your name : Jack
```

名称 Jack 实际上存储在 name1 中。将这个变量传递给函数 first_func()作为变量 username，在调用函数时，就会根据输入的参数做不同的响应，如代码 2.133 所示。

代码 2.133 函数参数示例

```
first_func(name1)
```

输出为

```
Hey Jack!
Jack, How do you do?
```

定义另一个函数 second_func()可进一步简化，该函数接受名称并将其存储在一个变量中，然后从函数内部调用 first_func()，如代码 2.134 所示。

代码 2.134 函数定义

```
def first_func(username):
    print("Hey", username + "!")
    print(username + "," ,"How do you do?")

def second_func():
    name = input("Please enter your name : ")
    first_func(name)

second_func()
```

输出为

```
Please enter your name : Tom
Hey Tom!
Tom, How do you do?
```

2.8.2 返回语句

当函数产生某个值且这个值需要返回给主算法以进行下一步操作时，可使用 return 语句，如代码 2.135 所示。

代码 2.135 函数返回

```
def times(x,y):
    z = x*y
    return z
```

上面定义的 times()函数接受两个参数并返回变量 z，该变量是两个参数的乘积，运行代码 2.136 可以查看效果。

代码 2.136　函数返回示例 1

```
c = times(4,5)
print(c)
```

输出为

```
20
```

函数计算的 z 值返回后存储在变量 c 中，可用于下一步操作。也可在 return 语句中直接使用表达式而不声明另一个变量，以便节省代码长度，如代码 2.137 所示。

代码 2.137　函数返回示例 2

```
def times(x,y):
    '''This multiplies the two input arguments'''
    return x*y
c = times(4,5)
print(c)
```

输出为

```
20
```

上述代码中写入了一行文本（或注释），在 help()函数中调用 times()函数时将返回该文本，如代码 2.138 所示。

代码 2.138　函数使用帮助

```
help(times)
```

输出为

```
Help on function times in module __main__:
times(x,y)
    This multiplies the two input arguments
```

函数也可以按照变量被定义的顺序返回多个变量的值，如代码 2.139 所示。

代码 2.139　函数返回多个变量的值 1

```
def multireturn_func(eglist):
    highest = max(eglist)
    lowest = min(eglist)
    first = eglist[0]
    last = eglist[-1]
    return highest,lowest,first,last
```

若调用函数时未为其分配任何变量，则结果在一个元组中返回，否则以 return 语句中声明的变量顺序输出结果，如代码 2.140 所示。

代码 2.140　函数返回多个变量的值 2

```
eglist = [10,50,30,12,6,8,100]
a = multireturn_func(eglist)
print(a)
a,b,c,d = egfunc(eglist)
print(' a  =',a,'\n b =',b,'\n c =',c,'\n d =',d)
```

输出为

```
(100, 6, 10, 100)
a = 100
b = 6
c = 10
d = 100
```

2.8.3 默认参数

当一个函数的参数在大多数情况下使用一个固定值时，可将这个值写到函数的参数列表中，如代码 2.141 所示。

代码 2.141 函数默认参数 1

```
01: def implicit_add(x, addnumber=3):
02:     return x+addnumber
```

implicit_add()接受两个参数，但大多数时候第一个参数只需加 3，因此第二个参数被默认赋值为 3。这里，第二个参数是隐式的，即第二个参数是有默认值的。现在，如果调用 implicit_add()函数时未定义第二个参数，那么第二个参数就默认为 3，如代码 2.142 所示。

代码 2.142 函数默认参数 2

```
01: implicitadd(4)
```

输出为

```
7
```

若定义了第二个参数值，则该值将覆盖分配给该参数的默认值，如代码 2.143 所示。

代码 2.143 函数默认参数 3

```
01: implicitadd(4, 4)
02: implicitadd(5, addnumber=6) # 建议采用这种调用方式，能够清楚地知道参数的含义
```

输出为

```
8
11
```

2.8.4 任意数量的参数

当函数接受的参数数量未知时，可在参数前使用星号，这样的变量是一个列表，它将所有参数都包含在列表中，如代码 2.144 所示。

代码 2.144 任意数量参数的函数

```
01: def add_n(*args):
02:     res = 0
03:     reslist = []
04:     for i in args:
05:         reslist.append(i)
06:     print(reslist)
07:     return sum(reslist)
```

上面的函数接受任意数量的参数，它定义一个列表，将所有参数包含到该列表中，并返回所有参数的总和，如代码 2.145 所示。

代码 2.145　任意数量参数的函数示例

```
01: add_n(1,2,3,4,5)
02: add_n(1,2,3)
```

输出为

```
[1, 2, 3, 4, 5]
15
[1, 2, 3]
6
```

与上例不同，下例展示如何使用变量名和值将任意数量的参数传入函数，如代码 2.146 所示。

代码 2.146　任意数量变量名和值的函数

```
01: def add_nd(**kwargs):
02:     res = 0
03:     reslist = []
04:     for (k,v) in kwargs.items():
05:         reslist.append(v)
06:     print(reslist)
07:     return sum(reslist)
08:
09: add_nd(x=10, y=20, c=30)
```

输出为

```
[10, 20, 30]
60
```

2.8.5　全局变量和局部变量

在 Python 语言中，函数内部声明的变量通常是局部变量，而函数外部声明变量的通常是全局变量。下面通过一个示例说明全局变量和局部变量的差别。在下面的函数中，向内部声明的列表中追加一个元素，函数内部声明的 eg1 是一个局部变量，如代码 2.147 所示。

代码 2.147　全局变量和局部变量示例

```
01: eg1 = [1,2,3,4,5]
02:
03: def egfunc1():
04:     eg1 = [1, 2, 3, 4, 5, 7]
05:     print("egfunc1>> eg1: ", eg1)
06:
07: def egfunc2():
08:     eg1.append(8)
09:     print("egfunc2>> eg1: ", eg1)
10:
11: def egfunc3():
12:     global eg1
13:     eg1 = [5, 4, 3, 2, 1]
14:     print("egfunc3>> eg1: ", eg1)
15:
16: egfunc1()
17: print("eg1: ", eg1, "\n")
```

```
18:
19: egfunc2()
20: print("eg1: ", eg1, "\n")
21:
22: egfunc3()
23: print("eg1: ", eg1, "\n")
```

输出为

```
egfunc1>> eg1: [1, 2, 3, 4, 5, 7]
eg1:  [1, 2, 3, 4, 5]

egfunc2>> eg1: [1, 2, 3, 4, 5, 8]
eg1:  [1, 2, 3, 4, 5, 8]

egfunc3>> eg1: [5, 4, 3, 2, 1]
eg1:  [5, 4, 3, 2, 1]
```

上述程序在 egfunc1 中给变量名 eg1 赋值，因此 eg1 是一个局部变量；在 egfunc2 中调用 eg1 对象的 append()函数，由于函数中未定义 eg1 对象，因此此处访问的是全局变量 eg1；在 egfunc3 中显式定义 eg1 为全局变量，因此访问的是全局变量。可以看出，如果是在局部空间中，那么局部变量可以覆盖全局变量的数据；一旦离开局部空间，局部变量就失效，全局变量就恢复到最初的值。

2.8.6 lambda 函数

程序中有时需要临时使用一个简单的函数，而单独定义这个函数则比较麻烦。为了提高编程效率，Python 等语言引入了 lambda 函数。lambda 函数不使用任何名称进行定义，只携带一个表达式，返回的是函数本身（类似于函数指针或函数对象）。这些函数由关键字 lambda 定义，后面跟变量、冒号和相应的表达式。lambda 函数在操作列表时非常方便，或者作为函数参数，如代码 2.148 所示。

代码 2.148 lambda 函数示例

```
01: z = lambda x: x * x
02: print(z(8))
03: print(type(z))
```

输出为

```
64
<class 'function'>
```

lambda 函数也可接受多个参数，如代码 2.149 所示。

代码 2.149 lambda 函数接受多个参数

```
01: zz = lambda x, y: (x*y, x**y)
02: zz(2, 3)
```

输出为

```
(6, 8)
```

2.9 类和对象

Python 中的变量、列表、字典等事实上是对象，对象是类的实例，而类是用来描述具有相同属性和方法的对象集，它定义集合中每个对象共有的属性和方法。面向对象编程具有质量高、效率高、

易扩展、易维护等优点，下面重点介绍“类”的概念。本书不涉及面向对象编程的理论部分[①]，仅介绍面向对象的概念及其在 Python 中的实现。

类声明如代码 2.150 所示。

代码 2.150　类声明

```
01: class ClassName:
02:     def functions(self, args):
03:         statements
```

在上面的代码中，functions 是类的成员函数。将一组与类关联的函数实现为类的成员函数，可以使程序的结构更清晰，并且便于对类进行扩展。

下面定义一个最简单的类——空类，如代码 2.151 所示。

代码 2.151　空类的定义

```
01: class FirstClass:
02:     pass
```

其中，pass 在 Python 中意味着什么都不做。

以上代码声明了一个名为 FirstClass 的类对象。下面考虑具有 FirstClass 的所有特征的变量 egclass。要做的是将 egclass 作为 FirstClass 的一个实例。在 Python 的术语中，这一操作称为创建实例，egclass 是 FirstClass 的实例，如代码 2.152 所示。

代码 2.152　类的实例化

```
01: egclass = FirstClass()
02: print(type(egclass))
03: print(type(FirstClass))
```

输出为

```
<class '__main__.FirstClass'>
<class 'type'>
```

2.9.1　成员函数与变量

类是现实事物的抽象表达，往往存在一些对内和对外的操作，即实现一些功能。这些功能一般代表类中的函数和变量，而类内的函数称为该类的“方法”，类中的变量称为属性。大多数类中都有一个名为__init__()的函数，它是类的构造方法，会在类实例化时自动调用。这个方法的作用一般是初始化类的变量，或者指定所有方法的初始化算法。例如，构造 FirstClass 接受两个变量名称和符号，可以更好地完成类的实例，生成特定功能的对象，如代码 2.153 所示。

代码 2.153　类的构造函数

```
01: class FirstClass:
02:     """My first class"""
03:     class_var = 10
04:     def __init__ (self, name, symbol):
05:         self.name = name
06:         self.symbol = symbol
```

① 面向对象编程的参考资料见https://zhuanlan.zhihu.com/p/338269391。

上面定义了一个函数，并且添加了__init__方法。下面创建一个 FirstClass 的实例，它接受两个参数，如代码 2.154 所示。

代码 2.154 类的实例化示例

```
01: eg1 = FirstClass('one', 1)
02: eg2 = FirstClass('two', 2)
03: print(eg1.name, eg1.symbol)
04: print(eg2.name, eg2.symbol)
05: print(eg1.__doc__)
```

输出为

```
one 1
two 2
My first class
```

可以看出，调用 FirstClass 时，它自动调用_ _init_ _函数，传入括号内的值，并将其赋值给类中的属性。

dir()函数在查看类包含什么及它提供什么方法时非常方便，如代码 2.155 所示。

代码 2.155 输出类的内容

```
01: dir(FirstClass)
```

输出为

```
['__class__',
'__delattr__',
'__dict__',
'__dir__',
...
'__str__',
'__subclasshook__',
'__weakref__',
'class_var']
```

实例的 dir()也能显示对象的属性和函数，如代码 2.156 所示。

代码 2.156 输出对象的内容

```
01: dir(eg1)
```

输出为

```
['__class__',
'__delattr__',
'__dict__',
'__dir__',
...
'__str__',
'__subclasshook__',
'__weakref__',
'class_var']
'name',
'symbol']
```

注意：self 不是 Python 内置的关键词，而由用户定义。可以使用任何名称，但使用 self 已成为默认写法，因此不建议替换成其他名称。代码 2.157 没有语法错误，但不建议这么写。

代码 2.157　类的构造函数

```
01: class FirstClass:
02:     def __init__(asdf1234,name,symbol):
03:         asdf1234.n = name
04:         asdf1234.s = symbol
05:
06: eg1 = FirstClass('one',1)
07: eg2 = FirstClass('two',2)
08: print(eg1.n, eg1.s)
09: print(eg2.n, eg2.s)
```

输出为

```
one 1
two 2
```

因为 eg1 和 eg2 是 FirstClass 的实例，所以不需要限制到 FirstClass 本身。它可以通过声明其他属性的方式来扩展自己，而不需要在 FirstClass 中声明属性，如代码 2.158 所示。

代码 2.158　对象的变量访问

```
01: eg1.cube = 1
02: eg2.cube = 8
03: dir(eg1)
```

输出为

```
['__class__',
'__delattr__',
...
'__str__',
'__subclasshook__',
'__weakref__',
'n',
's',
'cube']
```

就像全局变量和局部变量那样，类也有自己的变量类型。下面是类属性、实例属性的对比。

- 类属性：在方法外部定义的属性，适用于所有实例。
- 实例属性：在对象内部定义的属性，只适用于该对象，并且对每个实例都是独立的。

下面通过一个具体的例子来演示两种属性的差异，如代码 2.159 所示。

代码 2.159　类的变量访问

```
01: class FirstClass:
02:     test = 'test'
03:     def __init__(self,name,symbol):
04:         self.name = name
05:         self.symbol = symbol
```

这里 test 是一个类属性，而 name 是一个实例属性，如代码 2.160 所示。

代码 2.160 对象的变量访问

```
01: eg3 = FirstClass('Three', 3)
02: eg4 = FirstClass('Four', 4)
03: eg4.test = 'test4'
04: print(eg4.test)
05: print(eg3.test, eg3.name)
```

输出为

```
test4
test Three
```

2.9.2 继承

继承是面向对象编程的一种重要方式。通过继承，子类可以扩展父类的功能。父类是继承的类，也称基类；子类是从另一个类继承的类，也称派生类。

下面通过实例学习继承的用法。考虑 Person 类，它具有 salary 方法，如代码 2.161 所示。

代码 2.161 Person 类的设计

```
01: class Person:
02:     def __init__(self,name,age):
03:         self.name = name
04:         self.age = age
05:     def salary(self, value):
06:         self.money = value
07:         print(self.name,"earns",self.money)
08:
09: a = Person('Tom', 26)
10: a.salary(40000)
```

输出为

```
Tom earns 40000
```

下面考虑 Artist 类，它设置艺术家的薪水和艺术形式，如代码 2.162 所示。

代码 2.162 Artist 类的设计

```
01: class Artist:
02:     def __init__(self,name,age):
03:         self.name = name
04:         self.age = age
05:     def salary(self,value):
06:         self.money = value
07:         print(self.name, "earns", self.money)
08:     def artform(self, job):
09:         self.job = job
10:         print(self.name, "is a", self.job)
11:
12: b = Artist('Nitin', 20)
13: b.salary(50000)
14: b.artform('Musician')
```

输出为

```
Nitin earns 50000
Nitin is a Musician
```

可以发现两个类中 salary 方法的实现是一样的，这样重复实现同一功能会导致后续代码维护复杂等多种问题。为了消除代码重复，可在父类中实现重复的功能，而子类可以通过继承直接拥有父类已实现的函数。使用继承重新实现的 Artist 类如代码 2.163 所示。

代码 2.163　重新实现 Artist 类

```
01: class Artist(Person):
02:     def artform(self, job):
03:         self.job = job
04:         print(self.name,"is a", self.job)
05:
06: c = Artist('Nishanth',21)
07: c.salary(60000)
08: c.artform('Dancer')
```

输出为

```
Nishanth earns 60000
Nishanth is a Dancer
```

假设子类在继承一个特定的方法时，该方法不适合新类，那么可在新类中用相同的名称再次定义该方法来重写，如代码 2.164 所示。

代码 2.164　继承类的定义

```
01: class Artist(Person):
02:     def artform(self, job):
03:         self.job = job
04:         print(self.name, "is a", self.job)
05:
06:     def salary(self, value):
07:         self.money = value
08:         print(self.name,"earns",self.money)
09:         print("I am overriding the SoftwareEngineer class's salary method")
10:
11: c = Artist('Nishanth',21)
12: c.salary(60000)
13: c.artform('Dancer')
```

输出为

```
Nishanth earns 60000
I am overriding the SoftwareEngineer class's salary method
Nishanth is a Dancer
```

关于 Python 的类继承，需要注意的事项如下。

- 在继承中，基类的构造方法（__init__方法）不会被自动调用，它需要在其派生类的构造方法中专门调用。
- 在调用基类的方法时，需要加上基类的类名前缀，并且需要带上 self 参数变量。而在类中调用普通函数时，不需要带上 self 参数。

- Python 总是先查找对应类的方法，如果在派生类中找不到对应的方法，就开始到基类中逐个查找。

2.10 小结

通过本章的学习，读者应初步掌握 Python 的基本语法和用法[①]，但这还远远不够。本书仅介绍 Python 的基本用法，要想真正学懂 Python，还需要读者利用课后时间通过查阅一些资料（如廖雪峰的 Python 教程[②]）系统地学习 Python。和其他编程语言一样，需要进行大量的编程、调试、重构代码才能学好 Python，因此强烈建议读者完成本书配套的作业与习题。如果认为自己的编程能力比较弱，那么最好有意识地找一些练习题训练自己的编码能力，以将 Python 的用法记得更牢，发现学习 Python 的窍门。编写的代码越多，发现的窍门越多，就越欣赏这门语言。为了看懂并且编写实际中能用的程序，通常需要学习 Python 的第三方库。Python 框架、库和软件列表[③]总结了很多 Python 的第三方库，使用这些库可以加快解决问题的速度。

最后，享受解决问题的快乐吧！生命短暂，你需要 Python！

2.11 练习题

01. 统计单词出现的频数。给定一篇文章，找出每个单词的出现次数。例如，对代码 2.165 中的文本进行操作。当单词之间是两个空格或制表位时，程序是否有问题？如果出现“?”“/” 等与单词挨着的标点符号，应如何处理？

代码 2.165 示例文本

```
01: '''One is always on a strange road, watching strange scenery and
      listening to strange music. Then one day, you will find that the
      things you try hard to forget are already gone. '''
```

02. 无重复数字的三位数。有 1、2、3、4 个数字，能组成多少个互不相同且无重复数字的三位数？它们是多少？算法的复杂度是多少？

03. 判断。企业发放的奖金是根据利润提成的，具体规则如下。

- 利润低于或等于 10 万元时，按 10%提成。
- 利润大于 10 万元但低于 20 万元时，低于 10 万元的部分按 10%提成，高于 10 万元的部分按 7.5%提成。
- 利润在 20 万到 40 万之间时，高于 20 万元的部分按 5%提成。
- 利润在 40 万到 60 万之间时，高于 40 万元的部分按 3%提成。
- 利润在 60 万到 100 万之间时，高于 60 万元的部分按 1.5%提成。
- 利润高于 100 万元时，超过 100 万元的部分按 1%提成。

从键盘输入当月的利润 `I`，求应发放的奖金总数。除了用 `if` 来实现，能否用其他方式（如用 `list`，然后自动实现所有的判断）来实现？

① 本章的内容参考了 Python-Lectures，https://github.com/rajathkmp/Python-Lectures。

② 廖雪峰的 Python 教程：https://www.liaoxuefeng.com/wiki/1016959663602400。

③ http://awesome-python.com。

04. 乘法口诀表。输出 9×9 乘法口诀表。如何对齐，使得乘法口诀表看上去清楚？

05. while 循环。使用 while 循环输出 $2-3+4-5+\cdots+100$ 的和。除了直接法，能否用一句话写完？

06. 顺序排列。给出一个如代码 2.166 所示的数字列表，将其从大到小排序。

代码 2.166　示例数据

```
1, 10, 4, 2, 9, 2, 34, 5, 9, 8, 5, 0
```

07. 矩阵搜索。编写一个高效的算法搜索 $m \times n$ 矩阵 matrix 中的一个目标值 target，这个矩阵具有以下特性：每行的元素从左到右升序排列，每列的元素从上到下升序排列。例如，考虑代码 2.167 所示的矩阵。给定 target=5，返回 True；给定 target=20，返回 False。

代码 2.167　示例数据

```
[
[1, 4, 7, 11, 15],
[2, 5, 8, 12, 19],
[3, 6, 9, 16, 22],
[10, 13, 14, 17, 24],
[18, 21, 23, 26, 30]
]
```

08. 生成随机激活码。作为 Apple Store App 的独立开发者，你要开展限时促销活动，并为应用生成激活码（或优惠券）。使用 Python 如何生成 200 个激活码？例如，'KR603guyVvR' 是一个激活码。需要考虑什么是激活码？有什么特性？

09. 找到特定的文件。找到某个目录下某类型的所有文件。例如，找到 c:目录下的所有.dll 文件。注意，需要递归到每个目录中去查找。

10. 统计代码行数。假设你有一个目录，其中存储的是程序（假设是 C 或 Python 程序），统计写了多少行代码，包括空行和注释，但要分别列出（如 C 程序有多少行、Python 程序有多少行等）。

2.12　在线练习题

扫描如下二维码，访问在线练习题。