第2章 Python编程基础

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 人工智能基础与应用 | | 章名 | | | Python编程基础 | |
| 教学内容 | Python编程基础1 | | | | 课时 | 2 | |
| 项目性质 | □演示性 □验证性 □设计性 √综合性 | | | | | | |
| 授课班级 |  | 授课日期 | |  | | 授课地点 |  |
| 教学目标 | 了解Python入门  掌握Python开发环境搭建  掌握Python基础语法 | | | | | | |
| 教学内容 | 1. Python入门 2. Python开发环境搭建 3. Python基础语法 | | | | | | |
| 教学重点 | Python开发环境搭建、Python基础语法 | | | | | | |
| 教学难点 | Python开发环境搭建、Python基础语法 | | | | | | |
| 教学准备 | 装有Python的计算机  教学课件PPT  教材：《人工智能基础与应用（微课版）》 | | | | | | |
| 作业设计 |  | | | | | | |

教学过程

|  |  |
| --- | --- |
| **教学环节** | **教学内容与过程**  **（教学内容、教学方法、组织形式、教学手段）** |
| **课前组织** | 做好上课前的各项准备工作（打开计算机、打开课件、打开软件、打开授课计划、教案等），吸引学生注意力。 |
| **课程说明** | 【课前说明】  分别从Python语言的特点、Python开发环境搭建等知识点进行初步的了解。  【目的】  使学生从了解本节课的学习目标、学习重点、考评方式等方面明确课程学习的要求和目标。 |
| **课程内容描述** | 2.1 Python入门  Python是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。  1991年，第一个Python编译器诞生。它是由C语言实现的，且能够调用C语言的库文件。从其诞生，Python就已经具有了类、函数、异常处理、包含表和词典的核心数据类型，以及以模块为基础的拓展系统。Python语法很多来自C语言，但又受到荷兰国家数学与计算机科学研究中心设计的ABC语言强烈的影响。  Python具有很强的可读性，相对于其他语言经常使用英文关键字和英文标点符号，Python具有更有特色的语法结构。  Python是一种解释型语言：Python开发过程中没有编译环节，在这一点上其类似于PHP和Perl语言。  Python是交互式语言：Python可以在控制台中直接输入代码语句，按“Enter”键就可以直接执行该代码，而无须经过保存和编译阶段。  Python是面向对象语言：Python支持面向对象的或代码封装在对象的编程技术。  Python的优点很多，它自带很多库，如文件操作、网络通信、GUI编程、数据库处理等库，其中包含了大量的函数，因此使用者不必重复地编写某一类代码。同时除了这些内置的库外，Python还支持更多的第三方库，使用者可以直接导入并使用这些库，这就使得Python得到了更好的拓展。  本节课基于Python 3介绍Python的基础知识。  2.2 开发环境搭建  “工欲善其事，必先利其器”，搭建开发环境被誉为编程或者开发的开始，一个好的环境可以在编写代码时给出代码补全提示，有错误或者警告时也能够及时地给予显示。有一个稳定、易上手的开发环境对开发者而言是非常重要的，它能够帮助初学者更好地学习，帮助非初学者加快开发速度。  2.2.1 安装Python  Python可应用于多种平台，如Windows、Linux和Mac OS。  本书搭建基于Windows 64位的开发环境。如果要在Linux和Mac OS平台搭建Python开发环境，读者可下载源码或者可执行安装程序进行安装。Python 3.6.7的下载界面如图所示。    Python 3.6.7的下载界面  下载Python可执行安装程序后，双击进行安装。弹出“Python 3.6.7(64-bit) Setup”向导的“Install Python 3.6.7 (64-bit)”界面后，勾选“Add Python 3.6 to PATH”复选框，单击“Customize installation”选项，如图所示。    “Install Python 3.6.7(64-bit)”界面  接下来弹出“Optional Features”界面，单击“Next”按钮，如图所示。    “Optional Features”界面  在弹出的“Advanced Options”界面中，单击“Browse”按钮，选择Python安装路径，在“Customize install location”组合框中显示了当前Python的安装路径，单击“Install”按钮，等待安装完成，如图所示。    “Advanced Options”界面  Python安装结束后，出现“Setup was successful”界面，单击“Close”按钮关闭窗口，如图所示。  按“Win+R”组合键，打开“运行”对话框，在“打开”文本框中输入“cmd”，单击“确定”按钮，如图所示。    “Setup was successful”界面    “运行”对话框  此时出现“命令提示符”窗口，输入“python –V”，获得当前安装的Python版本，如图所示。    “命令提示符”窗口  直接在“命令提示符”窗口中输入“python”，可以进入Python交互模式，这个功能界面称为交互式解释器。在Python交互式解释器中可以直接输入Python的代码，无须经过保存和编译，按“Enter”键即可得到代码结果。直接输入“2+3”，并按“Enter”键，如果得到运算结果，则证明Python安装成功，如图所示。    在Python交互模式下验证Python是否安装成功  2.2.2 安装PyCharm  PyCharm是一种Python IDE，包含一整套可以帮助用户在使用Python语言进行开发时提高效率的工具，如调试、语法高亮、Project管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。此外，PyCharm还提供了一些高级功能，用于支持Django框架下的专业Web开发。  在PyCharm官网可以看到PyCharm具有专业版和社区版两个版本，专业版适用于Scientific和Web Python开发，支持HTML、JavaScript和SQL，可以免费试用；社区版可用于纯Python开发，是免费开源的，PyCharm的下载界面如图所示。    PyCharm的下载界面  下载PyCharm安装文件后，双击进行安装（这里针对社区版进行安装）。进入PyCharm社区版安装欢迎界面，单击“Next”按钮，如图所示。    PyCharm社区版安装欢迎界面  进入选择安装路径界面后，单击“Browse”按钮，选择安装路径，在“Destination Folder”组合框中显示当前PyCharm的安装路径，选择完毕后单击“Next”按钮，如图所示。    选择安装路径界面  进入安装选项界面，勾选“Create Desktop Shortcut”（创建桌面快捷方式）、“Update context menu”（更新上下文菜单）、“Create Associations”（.py文件默认用PyCharm打开）、“Update PATH variable (restart needed)”（更新路径变量，需要重启）中的复选框，单击“Next”按钮，如图所示。    安装选项界面  进入选择开始菜单文件夹界面，单击“Install”按钮，如图所示。    选择开始菜单文件夹界面  安装完成，进入完成PyCharm社区版安装界面，选中“Reboot now”单选按钮，单击“Finish”按钮，如图所示。    完成PyCharm社区版安装  等待系统重启，系统重启完成后双击PyCharm图标，进行配置，选择“Do not import settings”单选按钮，单击“OK”按钮，如图所示。    选择PyCharm配置界面  此时出现同意协议界面，勾选“I confirm that I have read and accept the terms of this User Agreement”复选框，单击“Continue”按钮，如图所示。    图2-16 同意协议界面  出现是否发送信息界面，单击“Send Usage Statistics”按钮，如图所示。    是否发送信息界面  进入UI主题选择界面，可以选择“Darcula”或“Light”单选按钮，这里选择“Light”单选按钮，单击“Next: Featured plugins”按钮，如图所示。    UI主题选择界面  进入其他插件界面，单击“Start using PyCharm”按钮完成PyCharm的设置，如图所示。    其他插件界面  2.2.3 体验PyCharm  在学习一门语言时，如C、C++，通常第一个代码就是输出“hello world”，这里Python的第一个代码也是输出“hello world”。  PyCharm初始化界面如图2-20所示，选择“Create New Project”选项。    PyCharm初始化界面  选择路径并单击“Create”按钮，“New Project”窗口如图所示。    “New Project”窗口  进入项目后，右键单击“basis”选项，在弹出的快捷菜单中选择“New”→“Python File”命令，如图所示。    选择“New”→“Python File”命令  在弹出对话框的“Name”文本框中输入“print”，单击“OK”按钮创建Python文件，如图所示。    “New Python file”对话框  在代码输入窗口中输入图2-24所示的代码。在该代码中，PyCharm会将保留关键字（保留关键字：不能用作常数或变量，也不能作为任何其他标识符名称，只包含小写）“print”变成蓝色，之后是一对括号和一对双引号。代码中的符号只有在英文输入法下输入才是正确的，用中文输入法输入符号后，程序会报错，双引号中的“hello world”是程序输出的内容。  Python中的保留关键字如表所示。  Python中的保留关键字   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | and | exec | not | assert | finally | or | | break | for | pass | class | from | print | | continue | global | raise | def | if | return | | del | import | try | elif | in | while | | else | is | with | except | lambda | yield |   Python的特点之一是依靠“Enter”键换行以及通过缩进来判断代码段，所以在编写Python代码时不需要添加分号，虽然添加分号后程序不会报错，但Python不提倡这种操作。图中的前两行为编码方式和注释，本书在后面的章节会对其进行讲解。这种编程模式为IDE下的编程模式。    输入代码  在执行代码之前，确定Python环境是否导入PyCharm，在PyCharm的菜单栏中选择“File”→“Settings”命令，如图所示。    选择“File”→“Settings”命令  进入“Settings”（设置）对话框，选择“Project: Python”→“Project Interpreter”选项，查看当前Python环境是否为安装的Python 3.6环境。如果不是，在“Project Interpreter”下拉列表中选择Python 3.6环境，设置完成的界面如图所示。    设置完成的界面  选择“Run”→“Run”命令，如图所示。  2-27  选择“Run”→“Run”命令  弹出“Run”对话框，选择“print”选项，如图所示。    “Run”对话框  在PyCharm的“Run”窗口中可以看到程序运行结果，如图所示。    程序运行结果  结果会输出“hello world”，“print”是Python中的输出保留关键字。  2.3 基础语法  在了解了Python的基础语法后可以更好地完成Python编程，由于本书在之后的章节中不会使用太过复杂的Python语法，所以本节主要介绍Python的常用基础语法。  2.3.1 基本输入和输出  输入和输出是学习代码的基础，有些开发者会使用输出当前代码的某些变量作为debug调试的一种方式。  1．基本输入  在Python 3中，input()函数会把用户输入的任何值都作为字符串来对待。下面介绍一个输入什么就输出什么的示例。  【例2-1】 在basis目录下新建文件，命名为input.py，在PyCharm中学习使用input()函数输入和使用print()函数输出。  variable = input()  print(variable)  在“Run”窗口中输入任意字符，如输入“hello world”，按“Enter”键，就会输出“hello world”，结果如图所示。    输入及输出结果  2．基本输出  在例2-1中使用print()函数完成输出时，直接将输出内容放在print()函数的括号中。接下来将“hello world”赋值给一个变量variable，然后进行输出。这里将之前的代码注释掉，注释方法是：将鼠标指针移到之前写的输出那一行，按下“Ctrl+/”组合键，完成注释，或者直接在那一行前面加上“#”进行注释。建议采用第一种方法，解除注释的方法是再次按下“Ctrl+/”组合键。  【例2-2】 将字符串赋值给一个变量，在PyCharm中编写代码。  variable = "hello world"  print(variable)  变量赋值输出结果如图所示。    变量赋值输出结果  variable是一个变量，创建时会在内存中开辟一块地址，“hello world”是一个字符串，可以由数字、字母、下划线组成，中间的“=”为赋值号，可将“=”右侧的值赋给左侧的值。  以上介绍的是直接输出一个字符串或者赋值后输出内容，接下来介绍格式化输出，即将数据按照指定的格式输出。  【例2-3】 将变量输出的代码注释，在PyCharm中写入格式化输出代码。  print("I want to say %s, you can say %s" %("hello world", "hello world"))  格式化输出的结果如图所示。    格式化输出的结果  在例2-3中，“%s”被称为%操作符，用来格式化输出字符串，使用其他的%操作符可以格式化输出其余类型，Python中的%操作符如表所示。  Python中的%操作符   |  |  | | --- | --- | | 符号 | 描述 | | %c | 格式化字符及其ASCII码 | | %s | 格式化字符串 | | %d | 格式化整数 | | %u | 格式化无符号整数 | | %o | 格式化无符号八进制数 | | %x | 格式化无符号十六进制数 | | %X | 格式化无符号十六进制数（大写） | | %f | 格式化浮点数字，可指定小数点后的位数 | | %e | 用科学计数法格式化浮点数 | | %E | 作用同%e，用科学计数法格式化浮点数 | | %g | %f和%e的简写 | | %G | %f和%e的简写 | | %p | 用十六进制数格式化变量的地址 |   要将“hello world”加双引号输出时，若直接在print()函数的引号内多加一对双引号，则程序会报错。  【例2-4】 直接在print()的引号内多加一对双引号，在PyCharm中尝试复现该错误。  print("I want to say "%s", you can say "%s"" %("hello world", "hello world"))  语法错误结果如图所示。    语法错误结果  如果要在字符串内部包含“”符号，需要用转义字符“\”来标识。  【例2-5】 在PyCharm中为字符串添加转义字符，输出引号。  print("I want to say \"%s\", you can say \"%s\"" %("hello world", "hello world"))  转义标识输出结果如图所示。    转义标识输出结果  从Python 2.6开始新增了一种格式化字符串的函数format()，它增强了字符串格式化的功能。通过“{}”“:”来代替以前的“%”。  【例2-6】 在PyCharm中用format()函数进行输出操作，将之前的代码注释掉。  # 不设置指定位置，按默认顺序  print("I want to say \"{} {}\", you can say \"{} {}\"".format("hello", "world", "hello", "world"))  在使用format()函数格式化时，print()函数输出的字符串中的“{}”会按format()函数中参数的默认顺序填充，每一个“{}”对应着一个参数，默认顺序输出结果如图所示。    默认顺序输出结果  【例2-7】 如果“{}”中有参数，则Python会按format()函数指定的位置填充并输出字符串，在PyCharm中尝试输入代码。  # 指定位置，按位置顺序输出  print("I want to say \"{0} {1}\", you can say \"{0} {1}\"".format("hello", "world"))  按指定位置输出的结果如图所示。    按指定位置输出的结果  2.3.2 Python运算符  运算符可针对一个以上的操作数项目进行运算操作，例如，2+3=5，2和3是操作数，+是运算符。  1．Python运算符类型  Python语言支持算术运算符、比较运算符、赋值运算符、位运算符、逻辑运算符、成员运算符、身份运算符。  （1）算术运算符  Python算术运算符及其描述如表所示。  Python算术运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | + | 加法运算 | | - | 减法运算 | | \* | 乘法运算 | | / | 除法运算 | | % | 取余运算 | | // | 取整运算（向下取整） | | \*\* | 乘幂运算 |   【例2-8】 在了解了算术运算符之后编写实例进行验证，在basis目录下新建文件，命名为arithmetic.py，学习Python算术运算符的操作，在PyCharm中编写代码。  operand\_a = 5  operand\_b = 2  operand\_c = 0  # 加法运算  operand\_c = operand\_a + operand\_b  print("{} + {} = {}".format(operand\_a, operand\_b, operand\_c))  # 减法运算  operand\_c = operand\_a - operand\_b  print("{} - {} = {}".format(operand\_a, operand\_b, operand\_c))  # 乘法运算  operand\_c = operand\_a \* operand\_b  print("{} \* {} = {}".format(operand\_a, operand\_b, operand\_c))  # 除法运算  operand\_c = operand\_a / operand\_b  print("{} / {} = {}".format(operand\_a, operand\_b, operand\_c))  # 取余运算  operand\_c = operand\_a % operand\_b  print("{} % {} = {}".format(operand\_a, operand\_b, operand\_c))  # 取整运算  operand\_c = operand\_a // operand\_b  print("{} // {} = {}".format(operand\_a, operand\_b, operand\_c))  # 乘幂运算  operand\_c = operand\_a \*\* operand\_b  print("{} \*\* {} = {}".format(operand\_a, operand\_b, operand\_c))  算术运算过程得到的结果如图所示。    算术运算过程得到的结果  （2）比较运算符  Python比较运算符及其描述如表所示。  Python比较运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | == | 等于号，比较是否相等，相等则返回True，否则返回False | | > | 大于号，比较前者是否大于后者，大于则返回True，否则返回False | | < | 小于号，比较前者是否小于后者，小于则返回True，否则返回False | | >= | 大于等于号，比较前者是否大于等于后者，大于等于则返回True，否则返回False | | <= | 小于等于号，比较前者是否小于等于后者，小于等于则返回True，否则返回False | | != | 不等于号，比较两者是否不相等，不等于则返回True，否则返回False |   在Python 2中，比较运算符“<>”的功能类似于“!=”，但该运算符在Python 3中被取消了。  【例2-9】 在了解了比较运算符之后编写实例进行验证，在basis目录下新建文件，命名为comparison.py，学习Python比较运算符的操作，在PyCharm中输入代码。  operand\_a = 5  operand\_b = 5  operand\_c = 2  # 等于号  print("{} == {} ? [{}]".format(operand\_a, operand\_b, operand\_a == operand\_b))  print("{} == {} ? [{}]".format(operand\_a, operand\_c, operand\_a == operand\_c))  # 大于号  print("{} > {} ? [{}]".format(operand\_a, operand\_b, operand\_a > operand\_b))  print("{} > {} ? [{}]".format(operand\_a, operand\_c, operand\_a > operand\_c))  # 小于号  print("{} < {} ? [{}]".format(operand\_a, operand\_b, operand\_a < operand\_b))  print("{} < {} ? [{}]".format(operand\_c, operand\_a, operand\_c < operand\_a))  # 大于等于号  print("{} >= {} ? [{}]".format(operand\_a, operand\_b, operand\_a >= operand\_b))  print("{} >= {} ? [{}]".format(operand\_c, operand\_a, operand\_c >= operand\_a))  # 小于等于号  print("{} <= {} ? [{}]".format(operand\_a, operand\_b, operand\_a <= operand\_b))  print("{} <= {} ? [{}]".format(operand\_a, operand\_c, operand\_a <= operand\_c))  # 不等于号  print("{} != {} ? [{}]".format(operand\_a, operand\_b, operand\_a != operand\_b))  print("{} != {} ? [{}]".format(operand\_a, operand\_c, operand\_a != operand\_c))  比较运算过程得到的结果如图所示。    比较运算过程得到的结果  （3）赋值运算符  Python赋值运算符及其描述如表所示。  Python赋值运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | = | 简单的赋值，右侧赋值给左侧 | | += | 加法赋值运算符，左侧加右侧赋值给左侧 | | -= | 减法赋值运算符，左侧减右侧赋值给左侧 | | \*= | 乘法赋值运算符，左侧乘右侧赋值给左侧 | | /= | 除法赋值运算符，左侧除以右侧赋值给左侧 | | %= | 取余赋值运算符，右侧对左侧取余赋值给左侧 | | //= | 取整赋值运算符，右侧对左侧取整赋值给左侧 | | \*\*= | 乘幂赋值运算符，左侧为底数，右侧为指数，将幂赋值给左侧 |   【例2-10】 在了解了赋值运算符之后编写实例进行验证，在basis目录下新建文件，命名为assignment.py，学习Python赋值运算符的操作，在PyCharm中输入代码。  operand\_a = 5  operand\_b = 2  operand\_c = 0  # 简单的赋值  operand\_c = operand\_a  print("{} = {}:".format(operand\_c, operand\_a))  print("{} = {}".format("operand\_c", operand\_c))  # 加法赋值  operand\_c = operand\_a  print("{} += {}:".format(operand\_c, operand\_b))  operand\_c += operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 减法赋值  operand\_c = operand\_a  print("{} -= {}:".format(operand\_c, operand\_b))  operand\_c -= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 乘法赋值  operand\_c = operand\_a  print("{} \*= {}:".format(operand\_c, operand\_b))  operand\_c \*= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 除法赋值  operand\_c = operand\_a  print("{} /= {}:".format(operand\_c, operand\_b))  operand\_c /= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 取余赋值  operand\_c = operand\_a  print("{} %= {}:".format(operand\_c, operand\_b))  operand\_c %= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 取整赋值  operand\_c = operand\_a  print("{} //= {}:".format(operand\_c, operand\_b))  operand\_c //= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  # 乘幂赋值  operand\_c = operand\_a  print("{} \*\*= {}:".format(operand\_c, operand\_b))  operand\_c \*\*= operand\_b  print("{} = {}".format("operand\_c", operand\_c))  赋值运算过程得到的结果如图所示。    赋值运算过程得到的结果  （4）位运算符  Python位运算符及其描述如表所示。  Python位运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | & | 按位与运算，二进制上的与运算，有0为0 | | | | 按位或运算，二进制上的或运算，有1为1 | | ～ | 按位取反运算，二进制上的取反 | | << | 左移，二进制上的左移 | | >> | 右移，二进制上的右移 | | ^ | 按位异或运算，二进制上的异或，相同为0 |   【例2-11】 在了解了位运算符之后编写实例进行验证，在basis目录下新建文件，命名为bit.py，学习Python位运算符的操作，在PyCharm中输入代码。  operand\_a = 5 # 5(DEC) == 0101(BIN)  operand\_b = 2 # 2(DEC) == 0010(BIN)  operand\_c = 0 # 0(DEC) == 0000(BIN)  # 按位与运算  operand\_c = operand\_a & operand\_b  print("{} & {} = {}".format(operand\_a, operand\_b, operand\_c))  # 按位或运算  operand\_c = operand\_a | operand\_b  print("{} | {} = {}".format(operand\_a, operand\_b, operand\_c))  # 按位取反运算  operand\_c = ～operand\_a  print("～{} = {}".format(operand\_a, operand\_c))  # 左移  operand\_c = operand\_a << 1  print("{} << 1 = {}".format(operand\_a, operand\_c))  # 右移  operand\_c = operand\_a >> 1  print("{} >> 1 = {}".format(operand\_a, operand\_c))  # 按位异或运算  operand\_c = operand\_a ^ operand\_b  print("{} ^ {} = {}".format(operand\_a, operand\_b, operand\_c))  位运算过程得到的结果如图所示。    位运算过程得到的结果  （5）逻辑运算符  Python逻辑运算符及其描述如表所示。  Python逻辑运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | and | 布尔与 | | or | 布尔或 | | not | 布尔非 |   布尔与、布尔或、布尔非不仅能对布尔值进行运算，还可以对其他类型进行运算。布尔与为x and y，如果x为True则返回y值，如果x为False则返回x值。布尔或为x or y，如果x为True则返回x值，如果x为False则返回y值。  【例2-12】 在了解了逻辑运算符之后编写实例进行验证，在basis目录下新建文件，命名为logic.py，学习Python逻辑运算符的操作，在PyCharm中输入代码。  operand\_a = 5  operand\_b = 2  operand\_c = 0  operand\_d = 0  # 布尔与  operand\_d = operand\_a and operand\_b  print("{} and {} = {}".format(operand\_a, operand\_b, operand\_d))  operand\_d = operand\_a and operand\_c  print("{} and {} = {}".format(operand\_a, operand\_c, operand\_d))  operand\_d = operand\_c and operand\_a  print("{} and {} = {}".format(operand\_c, operand\_a, operand\_d))  operand\_d = operand\_c and operand\_c  print("{} and {} = {}".format(operand\_c, operand\_c, operand\_d))  # 布尔或  operand\_d = operand\_a or operand\_b  print("{} or {} = {}".format(operand\_a, operand\_b, operand\_d))  operand\_d = operand\_a or operand\_c  print("{} or {} = {}".format(operand\_a, operand\_c, operand\_d))  operand\_d = operand\_c or operand\_a  print("{} or {} = {}".format(operand\_c, operand\_a, operand\_d))  operand\_d = operand\_c or operand\_c  print("{} or {} = {}".format(operand\_c, operand\_c, operand\_d))  # 布尔非  operand\_d = not operand\_a  print("not {} = {}".format(operand\_a, operand\_d))  operand\_d = not operand\_c  print("not {} = {}".format(operand\_c, operand\_d))  逻辑运算过程得到的结果如图所示。    逻辑运算过程得到的结果  （6）成员运算符  Python成员运算符及其描述如表所示。  Python成员运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | in | 在指定的序列中找到值则返回True，否则返回False | | not in | 在指定的序列中没有找到值则返回True，否则返回False |   【例2-13】 在了解了成员运算符之后，编写实例进行验证，在basis目录下新建文件，命名为member.py，学习Python成员运算符的操作，在PyCharm中输入代码。这个例子运用了列表的概念，这里只了解列表是有序的对象集合即可。  operand\_a = 5  operand\_b = 2  list = [1, 3, 5, 7]  print("{} in {} ? [{}]".format(operand\_a, list, operand\_a in list))  print("{} in {} ? [{}]".format(operand\_b, list, operand\_b in list))  成员运算过程得到的结果如图所示。    成员运算过程得到的结果  （7）身份运算符  Python身份运算符及其描述如表所示。  Python身份运算符及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | is | is用于判断两个标识符是不是引用自一个对象，如果是则返回True | | not is | not is用于判断两个标识符是不是引用自不同对象，如果是则返回True |   is与==是有区别的，is用于判断两个变量引用的对象是否为同一个（同一块内存空间），==用于判断引用变量的值是否相等，a is b相当于id(a)==id(b)，id()能够获取对象的内存地址。  【例2-14】 在了解了身份运算符之后编写实例进行验证，在basis目录下新建文件，命名为identity.py，学习Python身份运算符的操作，在PyCharm中输入代码。  operand\_a = 5  operand\_b = 5  operand\_c = 2  print("{} is {} ? [{}]".format(operand\_a, operand\_b, operand\_a is operand\_b))  print("{} is not {} ? [{}]".format(operand\_a, operand\_b, operand\_a is not  operand\_b))  print("{} is {} ? [{}]".format(operand\_a, operand\_c, operand\_a is operand\_c))  print("{} is not {} ? [{}]".format(operand\_a, operand\_c, operand\_a is not  operand\_c))  身份运算过程得到的结果如图所示。    身份运算过程得到的结果  以上例子都是在IDE模式下运行的，但是在交互模式下，为了提高内存利用效率，对于一些简单的对象，如一些数值较小的int对象，Python采取重用对象内存的方法。如a=5，b=5时，由于5为简单的int类型且数值小，所以Python不会为这两个变量分配两次内存，仅分配一次。但是当对象变大，例如，a=5000，b=5000时，如果对象定义时不在同一行，那么is的判断结果是False，但是在同一行定义对象时，is的判断结果就是True。当然，如果a和b是浮点型数据，则其比较结果与a=5000，b=5000的比较结果是一致的。  【例2-15】 Python交互模式下尝试is运算符的使用。  >>> a = 5  >>> b = 5  >>> a is b  True  >>> a = 5000  >>> b = 5000  >>> a is b  False  >>> a = 5000;b=5000  >>> a is b  True  >>> a = 5.0  >>> b = 5.0  >>> a is b  False  >>> a = 5.0;b = 5.0  >>> a is b  True  2．Python运算符优先级  一个表达式中可能包含多个不同运算符，由于该表达式有多种运算，所以不同的运算顺序可能得出不同的结果甚至出现错误的运算结果。当表达式中含有多种运算时，必须按一定顺序将其进行结合才能保证运算的合理性和结果的正确性、唯一性。Python运算符优先级及其描述如表所示，表中从上到下优先级依次降低。  Python运算符优先级及其描述   |  |  | | --- | --- | | 运算符 | 描述 | | \*\* | 指数（最高优先级） | | +、-、～ | 正负号、按位取反 | | \*、/、%、// | 乘、除、取余、取整除 | | +、- | 加法、减法 | | >>、<< | 右移、左移 | | & | 按位与 | | ^、| | 按位异或、按位或 | | <=、>=、<、> | 小于等于、大于等于、小于、大于 | | ==、!= | 等于、不等于 | | =、%=、/=、//=、-=、+=、\*=、\*\*= | 赋值运算符 | | is、is not | 身份运算符 | | in、not in | 成员运算符 | | not、and、or | 逻辑运算符 |   2.3.3 Python数据类型  在创建变量时，内存会开辟一块空间，可以存储整数、小数或字符，内存中的变量可以指定为不同的数据类型。  Python中的变量赋值与C语言是有区别的，它不需要int、double等关键字进行类型声明，但是每个变量在使用前都必须赋值，赋值以后该变量才会被创建。Python利用赋值运算符为变量赋值，其中，运算符左边是一个变量名，运算符右边是存储在变量中的内容。  1．数字  Python数字类型用于存储数值。  数字类型是不允许改变的，这就意味着如果改变数字类型的值，Python将重新分配一个对象。  Python支持4种不同的数字类型。  （1）有符号整型（int）：可以是正整数或负整数，不带小数点。  （2）长整型（long）：long类型只存在于Python 2.X版本中。在Python 2.2以后的版本中，int类型数据溢出后会自动转换为long类型。在Python 3.X版本中，long类型被int类型替代。  （3）浮点型（float）：浮点型由整数部分与小数部分组成，也可以使用科学计数法表示（2.5e2= 250）。  （4）复数（complex）：复数由实数部分和虚数部分构成，可以用a+bj或complex(a,b)表示。复数的实部a和虚部b都是浮点型。  2．字符串  为了方便查看结果，本小节使用Python交互模式编程。  在Python中，声明时使用双引号或者单引号的数据类型称为字符串，它是一种表示文本的数据类型。Python字符串有两种取值顺序：从左到右索引默认从0开始，索引的最大范围的值是字符串长度减1；从右到左索引是从-1开始的。  【例2-16】 实现Python字符串索引。  >>> string = "abcdef"  >>> string[0]  'a'  >>> string[-1]  'f'  使用“变量[头下标:尾下标]”的形式可以截取相应的字符串，实际取值范围是前闭后开的，其中，若头下标不填充，则默认从0开始算起。下标可以是正数或负数，代表从左向右索引还是从右向左索引，下标为空表示取到头或尾。字符串被截取后会返回一个新的对象，其中包含截取到的全部内容。  【例2-17】 实现Python字符串截取索引。  >>> string = "abcdef"  >>> string[0:6]  'abcdef'  >>> string[:-1]  'abcde'  >>> string[0:]  'abcdef'  加号“+”可以连接两个字符串，星号“\*”可以复制某个字符串，星号“\*”后的数字是复制次数。  【例2-18】 实现Python连接字符串以及复制字符串。  >>> string1 = "abc"  >>> string2 = "def"  >>> string1 + string2  'abcdef'  >>> string1 \* 2  'abcabc'  在字符串中可能会出现中文，但是计算机只能处理数字，所以就涉及编码问题。如果字符串中没有中文，则只需ASCII码就可以完成编码任务。由于加入中文后，ASCII的位数不够，所以我国制定了GB2312编码方式。使用该编码方式可实现中文编码，但是全世界的语言有很多种，每一种语言都有属于自己的编码方式，因此编码方式太多，太过混乱。基于此，Unicode诞生了，Unicode可将所有的语言统一到一套编码里。ASCII编码是一个字节，Unicode编码通常是两个字节，有些生僻字是4个字节，因此会出现一个问题，如果整个代码或者文本里面全部都是英文，而没有中文，那么内存上就需要多出一倍的存储空间，这对存储和传输都十分不利，所以要将Unicode编码进行升级，即出现了UTF-8编码。UTF-8编码是可变长的，英文是一个字节，一般中文是3个字节，生僻字是4～6个字节。使用UTF-8编码可以尽可能地节省内存，提高效率。所以在计算机内存中，统一使用Unicode编码，当编码内容保存在硬盘或者需要传输的时候，会转换为UTF-8。  由于.py文件也是一个文本文件，所以就需要将其保存为UTF-8编码。在解释器读取文件的时候，为了使其按UTF-8编码格式读取，通常在文件开头加上编码类型：  # -\*- coding: utf-8 -\*-  在PyCharm中，可以在软件右下方选择编码格式，如图所示。    选择编码格式  如果在文件开头加上编码类型，就需要对编码格式进行修改。  3．列表  列表（List）是Python中使用非常频繁的数据类型。列表可以完成大多数集合类数据结构的实现，列表中的元素可以是数字、字符串，甚至可以是列表（即嵌套）。列表用“[ ]”标识。  创建一个列表，只需要在方括号中添加元素并用逗号隔开即可。  【例2-19】 创建Python列表。  >>> list = [] # 创建空列表  >>> list = [1, 5, "aaa", [3, 4, "bbb"], 6]  >>> list  [1, 5, 'aaa', [3, 4, 'bbb'], 6]  列表中的每个元素都分配一个索引值，第一个元素索引是0，第二个元素索引是1，以此类推。列表可以进行索引、切片、加、乘、检查成员等操作。  【例2-20】 检索Python列表。  >>> list = [1, 5, "aaa", [3, 4, "bbb"], 6]  >>> list[0]  1  >>> list[2]  'aaa'  >>> list[3]  [3, 4, 'bbb']  >>> list[3][1]  4  列表中可以用变量[头下标:尾下标]的方式实现元素的切割，实际取值范围是前闭后开，从左到右索引默认从0开始，从右到左索引默认从-1开始，下标为空表示取到头或尾。  加号“+”可以连接两个列表，星号“\*”可以复制某个列表，星号“\*”后紧跟的数字是复制次数。  【例2-21】 Python列表截取索引。  >>> list = [1, 5, "aaa", [3, 4, "bbb"], 6]  >>> list[1:3]  [5, 'aaa']  >>> list[:-1]  [1, 5, 'aaa', [3, 4, 'bbb']]  >>> list[2:]  ['aaa', [3, 4, 'bbb'], 6]  >>> list \* 2  [1, 5, 'aaa', [3, 4, 'bbb'], 6, 1, 5, 'aaa', [3, 4, 'bbb'], 6]  列表中的元素通常使用del语句来删除。  【例2-22】 删除Python列表元素。  >>> list = [1, 5, "aaa", [3, 4, "bbb"], 6]  >>> list  [1, 5, 'aaa', [3, 4, 'bbb'], 6]  >>> del list[2]  >>> list  [1, 5, [3, 4, 'bbb'], 6]  4．元组  元组（Tuple）类似于列表（List），用“( )”标识，内部元素用逗号隔开。元组不能二次赋值，相当于只读的列表。  只需在括号中添加元素，并使用逗号隔开元素，即可创建元组。  【例2-23】 创建Python元组。  >>> tup = () # 创建空元组  >>> tup = (1,) # 元组中只有一个元素时，必须用逗号隔开，防止被作为括号运算  >>> tup  (1,)  >>> tup = (1, 5, "aaa", [3, 4, "bbb"], 6)  >>> tup  (1, 5, 'aaa', [3, 4, 'bbb'], 6)  【例2-24】 元组利用下标索引访问其中的元素。  >>> tup = (1, 5, "aaa", [3, 4, "bbb"], 6)  >>> tup  (1, 5, 'aaa', [3, 4, 'bbb'], 6)  >>> tup[1]  5  >>> tup[3]  [3, 4, 'bbb']  >>> tup[3][1]  4  【例2-25】 元组的元素值可以连接组合，但是不允许被修改，如果修改元组的元素值，程序会直接报错。  >>> tup = (1, 5, "aaa", [3, 4, "bbb"], 6)  >>> tup  (1, 5, 'aaa', [3, 4, 'bbb'], 6)  >>> tup \* 2  (1, 5, 'aaa', [3, 4, 'bbb'], 6, 1, 5, 'aaa', [3, 4, 'bbb'], 6)  >>> tup + tup  (1, 5, 'aaa', [3, 4, 'bbb'], 6, 1, 5, 'aaa', [3, 4, 'bbb'], 6)  >>> tup[0] = 3  Traceback (most recent call last):  File "<stdin>", line 1, in <module>  TypeError: 'tuple' object does not support item assignment  5．字典  字典（Dictionary）是无序的对象集合，用“{}”标识。字典由键（key）和它对应的值（value）组成。key可以是数字类型或字符串类型，value的数据类型不限，字典的每个key与value之间用冒号“:”隔开。  【例2-26】 字典可以先创建，后赋值。  >>> dict = {}  >>> dict["one"] = "this is one"  >>> dict[2] = 2  >>> dict  {'one': 'this is one', 2: 2}  【例2-27】 访问字典里面的值，需要将相应的键放入方括号。  >>> dict = {"one": "this is one", 2: 2, 3 : [1, 5, "aa"]}  >>> dict  {'one': 'this is one', 2: 2, 3: [1, 5, 'aa']}  >>> dict["one"]  'this is one'  >>> dict[3]  [1, 5, 'aa']  【例2-28】 为字典添加新内容的方法是增加新的键与其对应的值，修改字典内容的方法是为原有的键赋新的值。  >>> dict = {"one": "this is one", 2: 2, 3 : [1, 5, "aa"]}  >>> dict  {'one': 'this is one', 2: 2, 3: [1, 5, 'aa']}  >>> dict[2] = "this is two"  >>> dict  {'one': 'this is one', 2: 'this is two', 3: [1, 5, 'aa']}  >>> dict[4] = 4  >>> dict  {'one': 'this is one', 2: 'this is two', 3: [1, 5, 'aa'], 4: 4}  【例2-29】 可以删除字典中的某一对元素或者清空字典。  >>> dict = {"one": "this is one", 2: 2, 3 : [1, 5, "aa"]}  >>> dict  {'one': 'this is one', 2: 2, 3: [1, 5, 'aa']}  >>> del dict[2] # 删除键是2的条目  >>> dict  {'one': 'this is one', 3: [1, 5, 'aa']}  >>> dict  {'one': 'this is one', 3: [1, 5, 'aa']}  >>> dict.clear() # 清空字典所有条目  >>> dict  {}  >>> del dict # 删除字典  >>> dict  <class 'dict'>  6．数据类型转换  进行数据类型转换时，需要将转换后的数据类型作为函数名，数据类型转换函数及其说明如表所示。  数据类型转换函数及其说明   |  |  | | --- | --- | | 函数 | 说明 | | int(x [,base]) | 将x转换为一个整数 | | long(x [,base] ) | 将x转换为一个长整数 | | float(x) | 将x转换为一个浮点数 | | complex(real [,imag]) | 创建一个复数 | | str(x) | 将对象x转换为字符串 | | repr(x) | 将对象x转换为表达式字符串 | | eval(str) | 用来计算字符串中的有效Python表达式，并返回一个对象 | | tuple(s) | 将序列s转换为一个元组 | | list(s) | 将序列s转换为一个列表 | | set(s) | 将s转换为可变集合 | | dict(d) | 创建一个字典，d必须是一个序列(key,value)元组 | | frozenset(s) | 将s转换为不可变集合 | | chr(x) | 将一个整数转换为字符 | | unichr(x) | 将一个整数转换为Unicode字符 | | ord(x) | 将一个字符转换为它的整数值 | | hex(x) | 将一个整数转换为十六进制字符串 | | oct(x) | 将一个整数转换为八进制字符串 |   2.3.4 Python语句  Python语句大致可分为条件语句和循环语句两种。  1．条件语句  为了方便查看结果，本小节均使用Python交互模式编程。  Python条件语句通过一条或多条语句的执行结果（True或False）来决定是否执行代码块。  Python与其他语言的区别之一是Python不使用大括号“{}”来控制代码块范围，而是使用缩进限定代码块，缩进的空白数量是可变的，但是同一个代码块内的语句必须包含相同的缩进。非同一个代码块的缩进量可以不同，但建议整个项目使用相同的缩进量，以便于阅读。  【例2-30】 尝试Python的缩进规则。  >>> if True:  ... print("True")  Python语言规定：任何非0和非空（Null）值为True，0或者空值为False。条件语句的执行过程如图所示。  2-45  条件语句的执行过程  Python中if语句用于控制代码块是否执行，基本形式为：  if 判断条件语句：  执行语句1  else:  执行语句2  其中，“判断条件语句”成立（为真）时，执行后面的语句，执行语句可以为单行或者多行，用缩进来区分是否为同一代码块。else为可选语句，当“判断条件语句”不成立时，执行else后的语句。  【例2-31】 学习Python中if…else的使用。  >>> a = True  >>> if a:  ... print("a is true")  ... else:  ... print("a is false")  ...  a is true  除了if…else之外，还有if…elif…else形式，其基本形式为：  if 判断条件语句1:  执行语句1  elif 判断条件语句2:  执行语句2  else:  执行语句3  当满足“判断条件语句1”时，执行语句1，然后整个if语句结束。  当不满足“判断条件语句1”时，判断是否满足“判断条件语句2”，如果满足，则执行语句2，然后整个if语句结束。  当不满足“判断条件语句1”“判断条件语句2”时，执行语句3，然后整个if语句结束。  【例2-32】 学习Python中if…elif…else的使用。  >>> a = 23  >>> if a < 10:  ... print("a < 10")  ... elif a < 20:  ... print("a < 20")  ... else:  ... print("a >= 20")  ...  a >= 20  if语句还可以嵌套使用，基本形式为：  if 判断条件语句1：  执行语句1  if 判断条件语句2：  执行语句2  else:  执行语句3  else:  执行语句4  【例2-33】 Python中if语句嵌套的具体使用方法。  >>> a = 23  >>> if a > 10:  ... if a > 20:  ... print("a > 20")  ... else:  ... print("10 < a < 20")  ... else:  ... print("a <= 10")  ...  a > 20  2．循环语句  需要多次重复执行的代码都可以用循环的方式来完成。循环语句可以提高代码的重复使用率，为开发者提供方便。循环语句的执行过程如图所示。  2-46  循环语句的执行过程  Python提供了for循环和while循环（Python中没有do…while循环）。  Python编程中，while语句用于循环执行程序，即在某条件下循环执行某段程序，用来完成需要重复执行的任务。  while循环基本形式为：  while 条件:  执行语句1  当while条件为真时，执行语句1，直到while条件为假，跳出循环。  【例2-34】 学习Python中的while循环。  >>> i = 0  >>> while i < 5:  ... print(i)  ... i += 1  ...  0  1  2  3  4  当i小于5的时候，会打印输出i，然后让i加1，并重新赋值给i；直到i大于或等于5时，该循环条件不满足，循环结束。  循环语句还有两个重要的命令：continue和break。continue用于跳过本次循环，break用于退出循环。此外，“条件”可以是一个常量，表示循环必定成立，除非遇到跳出语句，否则就一直循环，这称为“死循环”。  【例2-35】 学习Python中continue、break的具体使用。  >>> i = 0  >>> while True:  ... i += 1  ... if i > 6:  ... break  ... else:  ... if i % 2:  ... print(i)  ... else:  ... continue  ...  1  3  5  这里利用“死循环”让代码块中的程序一直循环。当i>6时退出循环，当i为奇数时将其输出，否则就跳过本次循环，进行下次循环。  Python中除了while循环外还有for循环，for循环可以遍历任何序列的内容，如一个列表或者一个字符串。  for循环的基本形式为：  for元素in序列:  执行语句1  使用for循环遍历列表或者字符串时，需要配合成员运算符in完成。  【例2-36】 利用for循环遍历字符串和列表。  >>> for letter in "Python":  ... print(letter)  ...  P  y  t  h  o  n  >>> for letter in [1, "a"]:  ... print(letter)  ...  1  a  for循环不仅可以通过元素来遍历，还可以通过下角标来遍历。  【例2-37】 利用for循环，通过下角标遍历字符串和列表。  a = "Python"  >>> for i in range(len(a)):  ... print(a[i])  ...  P  y  t  h  o  n  >>> a = [1, "a"]  >>> for i in range(len(a)):  ... print(a[i])  ...  1  a  2.3.5 Python函数  函数是由具有独立功能的代码块组成的一个模块，函数能提高代码的模块性和重复利用率。Python提供了许多内置函数，如print()。开发者自己可以创建函数，自己创建的函数称为用户自定义函数。  定义函数格式：  def 函数名():  代码  使用函数名可实现调用。  【例2-38】 完成简单的函数并调用。  >>> def caculate():  ... a = 1  ... b = 2  ... c = a + b  ... print(c)  ...  >>> caculate()  3  1．带参数的函数  带参数的函数定义时需要在函数小括号内添加多个参数名称，调用时需要在小括号内加入要传递的参数的值。  基本格式为：  def 函数名(参数名1，参数名2，...) :  函数执行代码  【例2-39】 完成一个带参数的函数，功能为将两个参数相加并调用。  >>> def caculate(a, b):  ... c = a + b  ... print(c)  ...  >>> caculate(1, 2)  3  参数a对应1，b对应2，在调用函数时将参数传递进去，得到的c就是1与2之和。  当调用函数时，参数的值如果没有传入，则被认为是默认值。带有默认值的参数一定要位于参数列表的最后面。  当一个函数要处理比声明时更多的参数时，这些待处理的参数称为不定长参数，声明时不会命名。  含有不定长参数的函数格式为：  def 函数名(参数名1,\*args,\*\*kwarg)：  函数执行代码  args是元组，加“\*”表示args可以存放所有未命名的变量参数；kwarg是字典，加“\*\*”表示kwarg可以存放命名参数，如key=value的参数。  2．带返回值的函数  要使函数返回结果给调用者，需要在函数中使用return语句。  带返回值的函数的基本格式为：  def 函数名(参数名1，参数名2，...) :  函数执行代码  return 结果值  【例2-40】 完成一个带返回值的函数并调用。  >>> def caculate(a, b):  ... c = a + b  ... return c  ...  >>> d = caculate(1, 2)  >>> print(d)  3  返回多个值时，需要赋值给多个变量。 |
| **总结评价** | 本节课主要对Python语言进行了简单的学习，包括Python开发环境搭建、Python的基本语法等。 |