第一单元：初识 Python 的世界

Python 的创始人为荷兰的 Guido，他在 1982 年获得了阿姆斯特丹大学的数学和计算机科学专业所示学位，1989 年，为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，做为 ABC 语言的一种继承，于是开始编写 Python 语言的编译器。之所以选中 Python 作为该编程语言的名字，是因为他是一个叫 Monty Python 的喜剧团体的爱好者。

Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。Python 的最大特点是拥有一个广泛活跃的科学计算社区，从而为解决 Python 使用过程遇到的各类问题提供了有力的保障。

1.1 Python 发展历程

Python 本身也是由诸多其他语言发展而来的,这包括 ABC、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell 和其他的脚本语言等等。像 Perl 语言一样，Python 源代码同样遵循 GPL(GNU General Public License)协议。

现在 Python 是由一个核心开发团队在维护，Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

版本号	发布时间	拥有者	GPL 兼容
0.9.0~1.2	1991~1995	CWI	是
1.3~1.5.2	1995~1999	CNRI	是
1.6	2000	CNRI	否
2.0	2000	BeOpen.com	否
1.6.1	2001	CNRI	否
2.1	2001	PSF	否
2.0.1	2001-06-22	PSF	是
2.2~2.7.11	2001~2015	PSF	是
2.7.12	2016-06	PSF	是
2.7.13	2016-12	PSF	是
3.x	2008~至今	PSF	是

各个时期的版本

2014 年 11 月，Python2.7 将在 2020 年停止支持的消息被发布，并且不会在发布 2.8 版本，建议用户尽可能的迁移到 3.x+。

Python 最初发布时，在设计上有一些缺陷，比如 Unicode 标准晚于 Python 出现，所以一直以来对 Unicode 的支持并不完全，而 ASCII 编码支持的字符有限。

Python3 相对 Python 早期的版本是一个较大的升级，Python 3 在设计的时候没有考虑向下兼容，所以很多早期版本的 Python 的程序无法再 Py3 上运行。

为了照顾早期的版本，推出过渡版本 2.6——基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。

2010 年继续推出了兼容版本 2.7，大量 Python3 的特性被反向

迁移到了 Python2.7。2.7 比 2.6 进步非常多，同时拥有大量 3 中的特性和库，并且照顾了原有的 Python 开发人群。

1.2 Python 语言特点及其应用

Python 语言受到如此多开发人员的青睐，主要是具有如下这些特点：

1.易于学习：Python 有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。它使我们能专注于解决问题而不是去明白语言本身。

2.免费且开源：Python 是一种开源语言，其源代码是自由开放的。我们可以自由的发布这个软件的拷贝，阅读她的源代码，对它做改动，把它的一部分用于新的自由软件中。

3.易于维护：Python 的成功在于它的源代码是相当容易维护的。

4.一个广泛的标准库：Python 的最大的优势之一是丰富的库，跨平台的，在 UNIX，Windows 和 Macintosh 兼容很好。

5.互动模式：互动模式的支持，您可以从终端输入执行代码并获得结果的语言，互动的测试和调试代码片断。

6.可移植：基于其开放源代码的特性，Python 已经被移植（也就是使其工作）到许多平台。

7.可扩展：如果你需要一段运行很快的关键代码，或者是想要编写一些不愿开放的算法，你可以使用 C 或 C++完成那部分程序，然后从你的 Python 程序中调用。

8.面向对象：Python 既支持面向过程的编程也支持面向对象的编

程。与其他主要的语言如 C++ 和 Java 相比，Python 以一种非常强大又简单的方式实现面向对象编程。

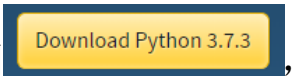
9.GUI 编程：Python 支持 GUI 可以创建和移植到许多系统调用。

10.可嵌入：可以将 Python 嵌入到 C/C++程序，也可以将 CC/C++程序嵌入到 Python，让用户获得"脚本化"的能力。

1.3 Python 开发环境的下载与配置

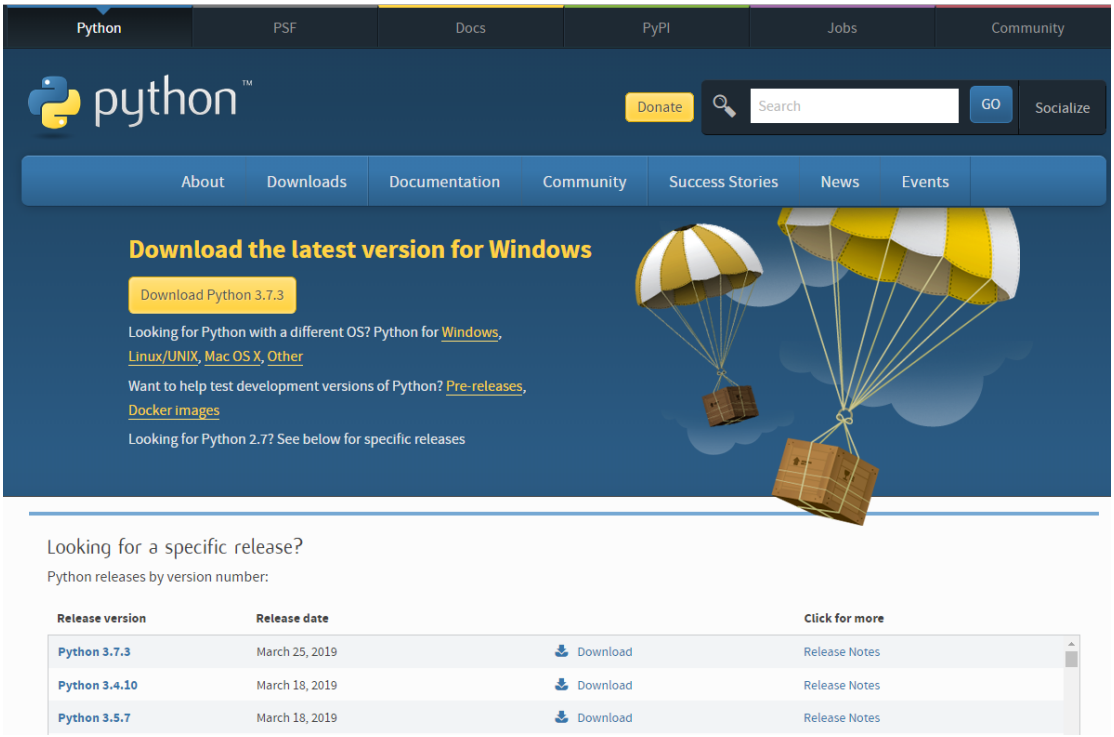
不同的设备与不同的系统都可以快速获得 Python，即使是手机，也可以体验 Python。

1.3.1Window 安装 Python

本书将以 Windows 为开发平台，在 Windows 中打开浏览器，访问 Python 的官方网站 http://www.python.org/download/，如下图所示，将鼠标移动到 

，本书以 Python 3 为基础。


扫码看视频 1.1

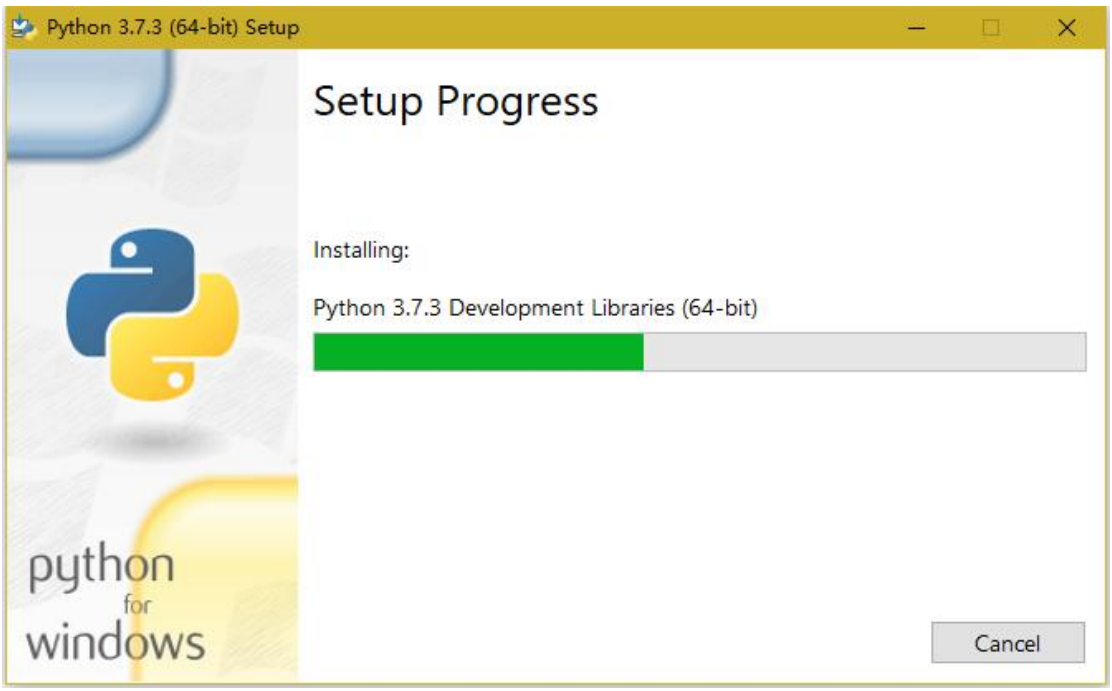


Python 官方网站下载页面

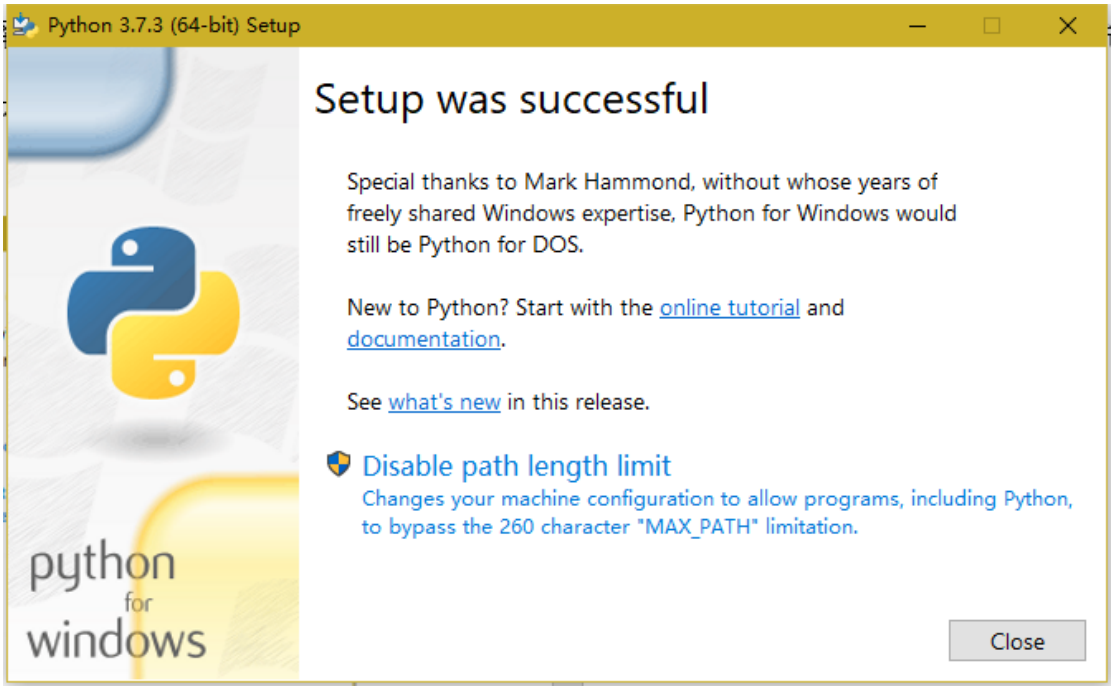
如果需要其它版本，可以在上图中选择自己需要的版本进行下载。接下来就以 Python3.7.3 的安装为例。下图是 Python 安装器启动后的界面。

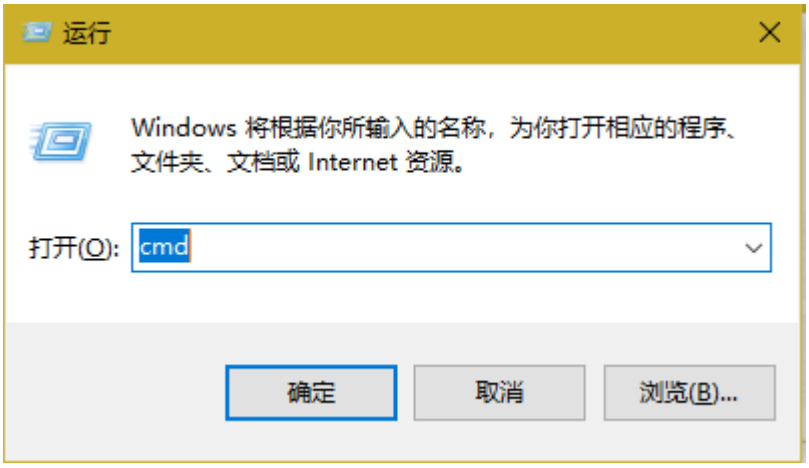
这里请勾选“Add Python 3.7 to PATH”，以便之后直接在命令行中使用，

然后下图是安装过程。

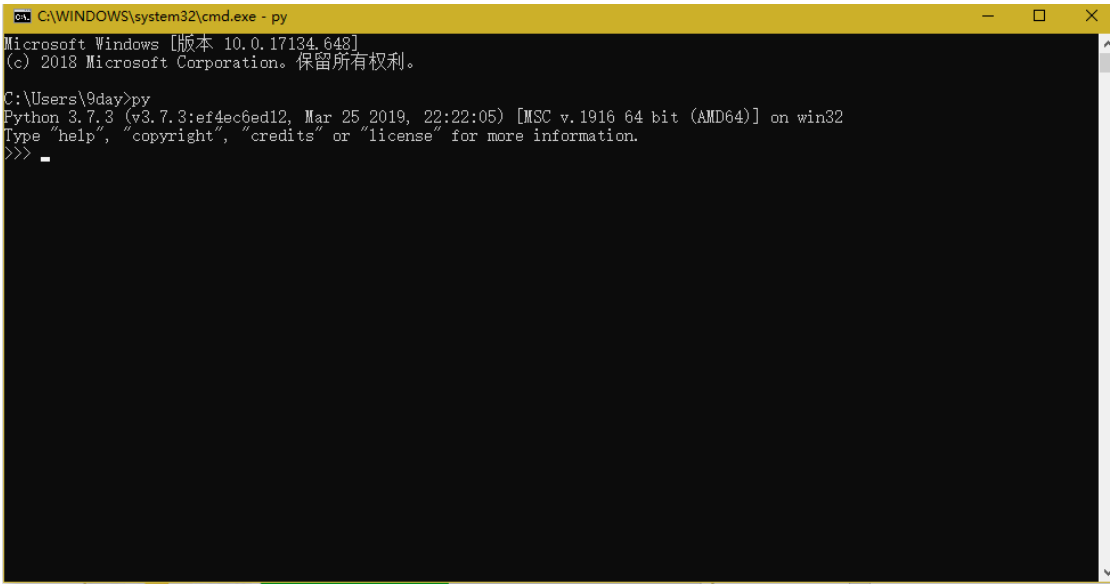


直到出现下图界面说明安装完成了。



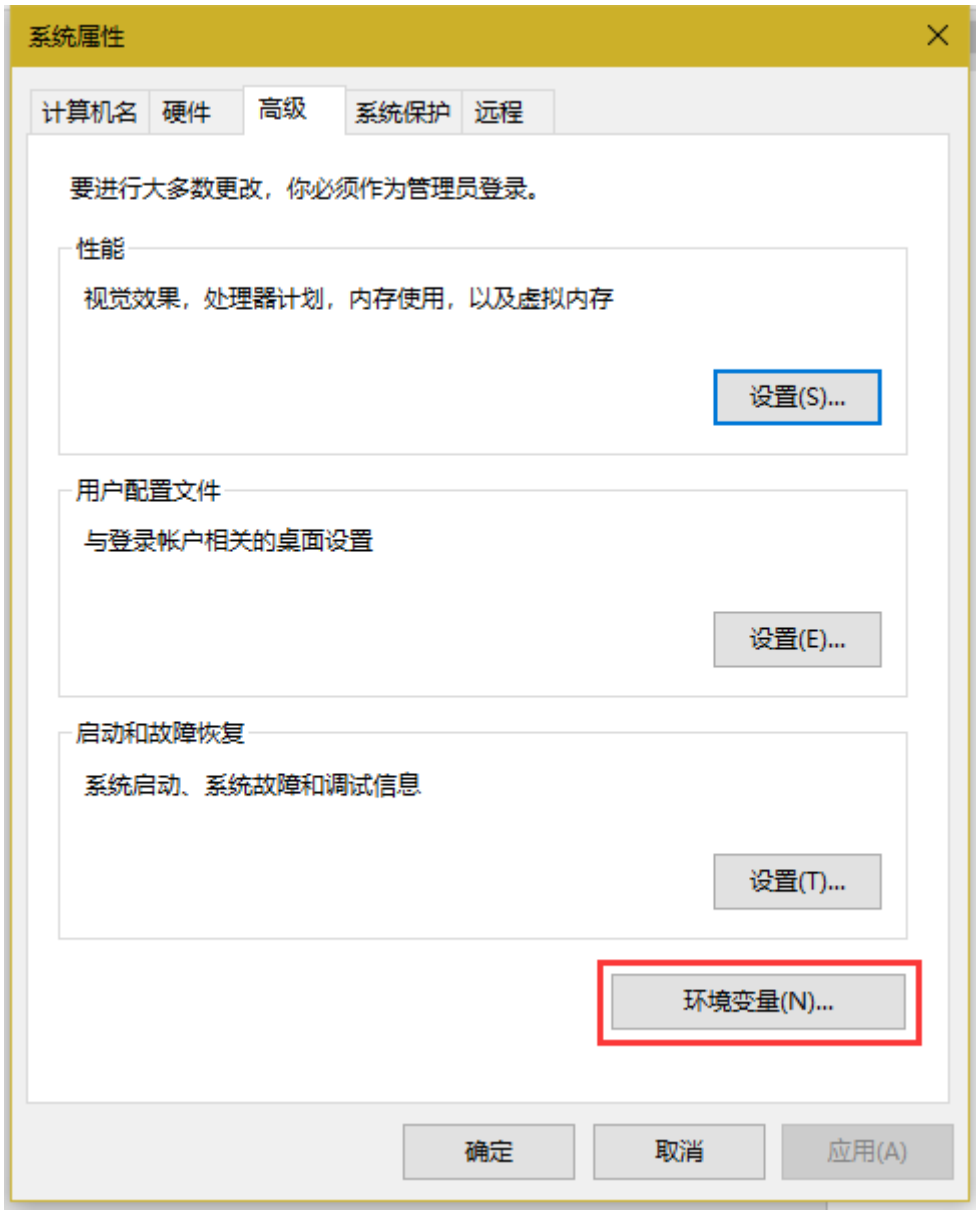


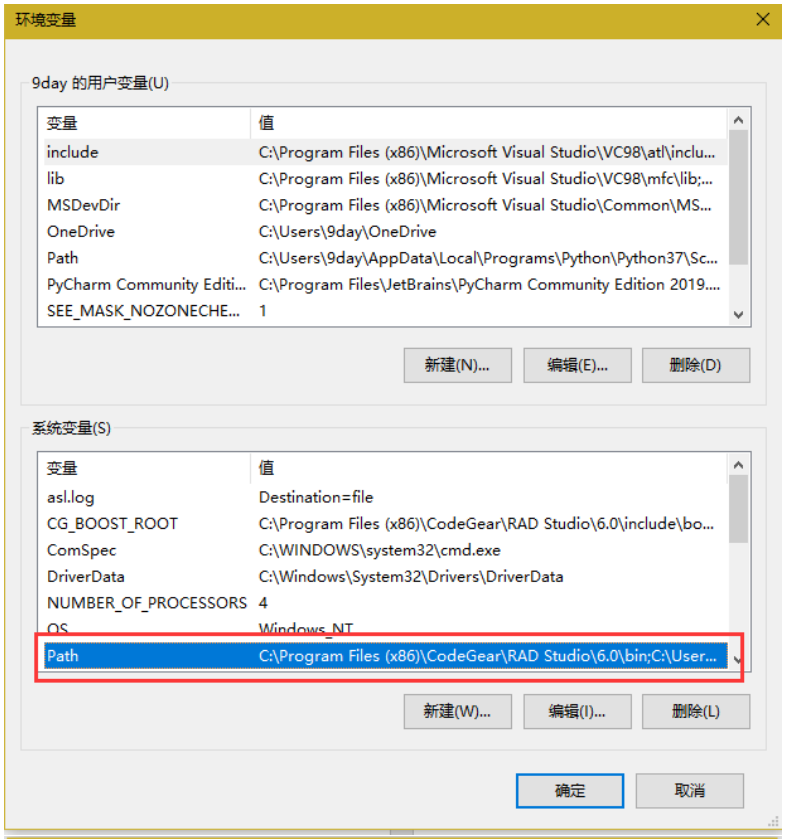
使用组合件 win+R 调出运行窗口，输入 cmd，回车启动命令行窗口，在窗口中输入 py 检查是否正确安装。

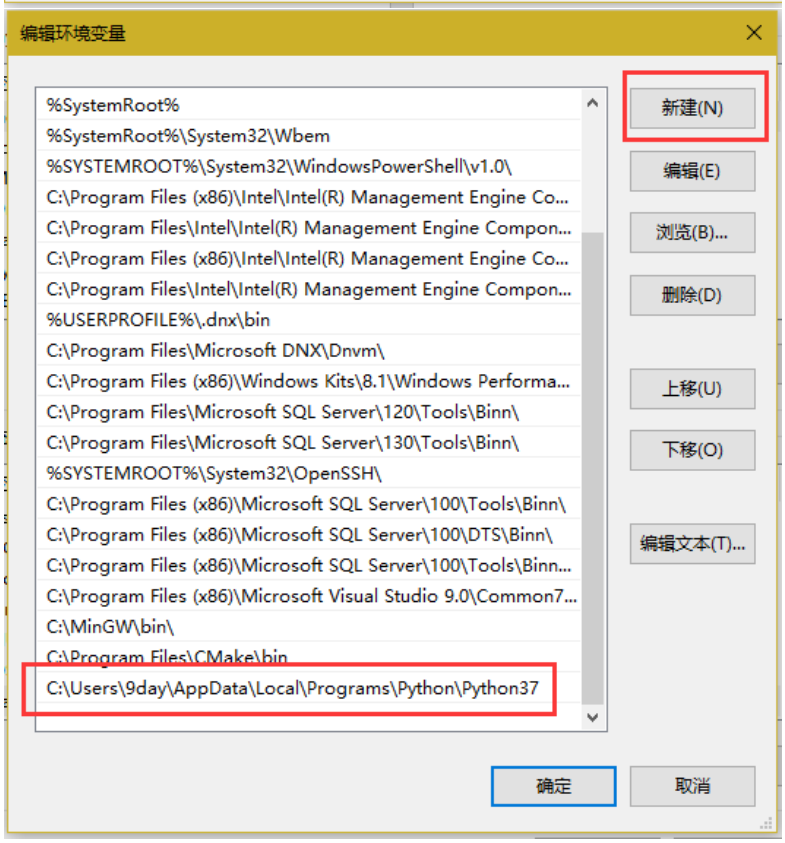


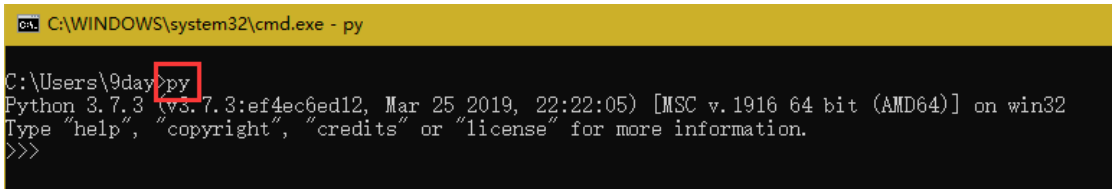
若出现上述提示则说明已安装成功。若失败需要进行环境变量配置，鼠标右键选择桌面“计算机”的“属性”，打开系统属性。


系统
控制面板 › 系统和安全 › 系统
搜索控制面板
控制面板主页
设备管理器
远程设置
系统保护
高级系统设置
查看有关计算机的基本信息
Windows 版本
Windows 10 教育版
© 2018 Microsoft Corporation。保留所有权利。
Windows 10
系统
处理器: Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.41 GHz
已安装的内存(RAM): 4.00 GB (3.87 GB 可用)
系统类型: 64 位操作系统，基于 x64 的处理器
笔和触控: 没有可用于此显示器的笔或触控输入
计算机名、域和工作组设置
计算机名: DESKTOP-QMM8CL3
更改设置
计算机全名: DESKTOP-QMM8CL3
计算机描述:
工作组: WORKGROUP
Windows 激活
Windows 已激活 阅读 Microsoft 软件许可条款
产品 ID: 00328-10000-00001-AA933
更改产品密钥
另请参阅
安全和维护


系统属性
计算机名
硬件
高级
系统保护
远程
要进行大多数更改，你必须作为管理员登录。
性能
视觉效果，处理器计划，内存使用，以及虚拟内存
设置(S)...
用户配置文件
与登录帐户相关的桌面设置
设置(E)...
启动和故障恢复
系统启动、系统故障和调试信息
设置(T)...
环境变量(N)...
确定
取消
应用(A)


环境变量
9day 的用户变量(U)
变量
值
include
C:\Program Files (x86)\Microsoft Visual Studio\VC98\atl\inclu...
lib
C:\Program Files (x86)\Microsoft Visual Studio\VC98\mfc\lib;...
MSDevDir
C:\Program Files (x86)\Microsoft Visual Studio\Common\MS...
OneDrive
C:\Users\9day\OneDrive
Path
C:\Users\9day\AppData\Local\Programs\Python\Python37\Sc...
PyCharm Community Editi...
C:\Program Files\JetBrains\PyCharm Community Edition 2019....
SEE_MASK_NOZONECHE...
1
新建(N)...
编辑(E)...
删除(D)
系统变量(S)
变量
值
asl.log
Destination=file
CG_BOOST_ROOT
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\include\bo...
ComSpec
C:\WINDOWS\system32\cmd.exe
DriverData
C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS
4
OS
Windows_NT
Path
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\bin;C:\User...
新建(W)...
编辑(I)...
删除(L)
确定
取消

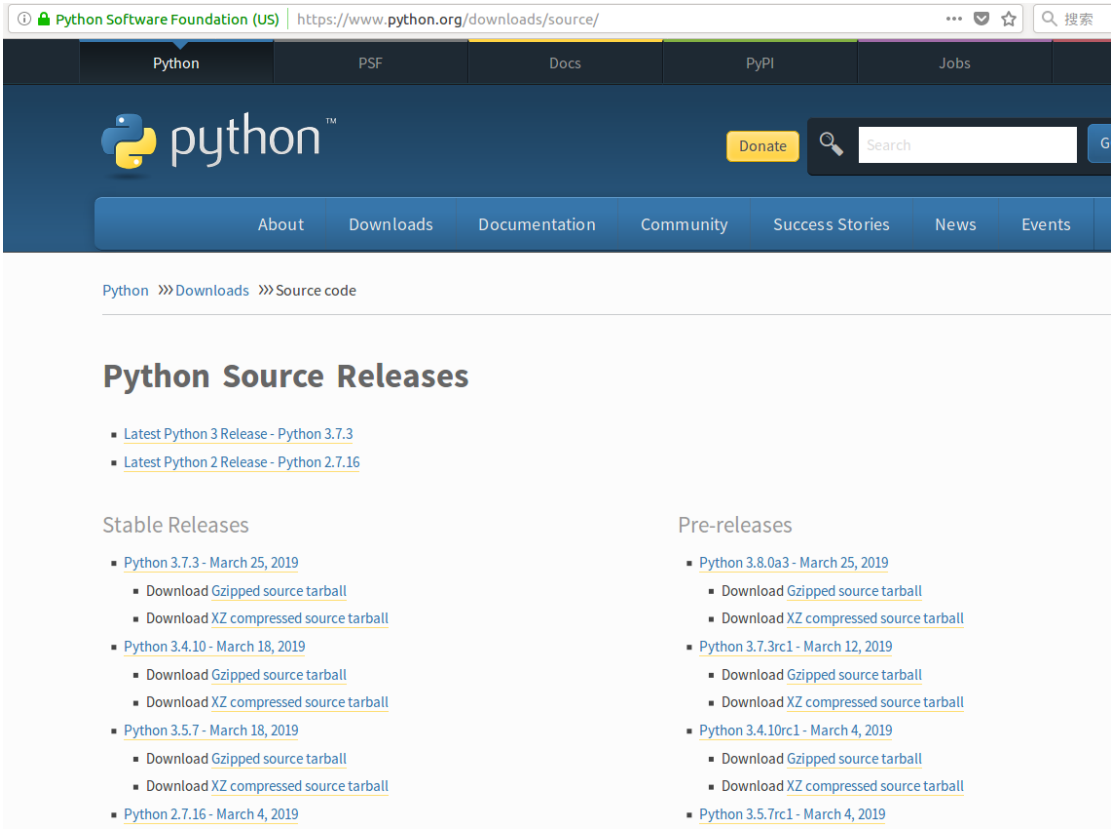

编辑环境变量
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
C:\Program Files (x86)\Intel\Intel(R) Management Engine Co...
C:\Program Files\Intel\Intel(R) Management Engine Compon...
C:\Program Files (x86)\Intel\Intel(R) Management Engine Co...
C:\Program Files\Intel\Intel(R) Management Engine Compon...
%USERPROFILE%\.dnx\bin
C:\Program Files\Microsoft DNX\Dnvm\
C:\Program Files (x86)\Windows Kits\8.1\Windows Performa...
C:\Program Files\Microsoft SQL Server\120\Tools\Binn\
C:\Program Files\Microsoft SQL Server\130\Tools\Binn\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn\
C:\Program Files (x86)\Microsoft SQL Server\100\DTS\Binn\
C:\Program Files (x86)\Microsoft SQL Server\100\Tools\Binn...
C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7...
C:\MinGW\bin\
C:\Program Files\CMake\bin
C:\Users\9day\AppData\Local\Programs\Python\Python37
新建(N)
编辑(E)
浏览(B)...
删除(D)
上移(U)
下移(O)
编辑文本(T)...
确定
取消


C:\WINDOWS\system32\cmd.exe - py
C:\Users\9day>py
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

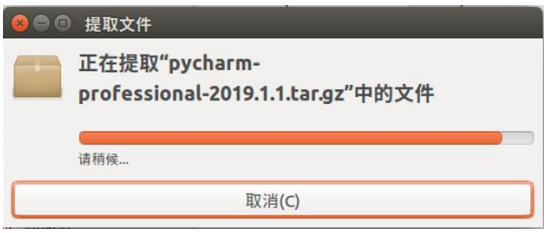
1.3.2 Linux 安装 Python

实际上在 Ubuntu16.04 版本中，系统是自带了 python2.7,如果需要使用 python3 则需要使用如下方式进行配置开发环境。

1. 进入官网下载（https://www.python.org/downloads/source/）



2. 将下载下来的 Python-3.7.3.tgz 压缩包解压至当前目录



3. 在安装之前，使用 Ctrl+Alt+T 快捷键打开终端，使用以下命令安装 Python 的先决条件。

```
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
sudo apt-get install libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
```


day@day-OptiPlex-7050: ~
[1]+ 已停止 python
day@day-OptiPlex-7050:~$ sudo apt-get install build-essential checkinstall
[sudo] day 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
build-essential 已经是最新版 (12.1ubuntu2)。
下列【新】软件包将被安装：
checkinstall
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 559 个软件包未被升级。
需要下载 121 kB 的归档。
解压缩后会消耗 516 kB 的额外空间。
您希望继续执行吗？ [Y/n] y
获取:1 http://cn.archive.ubuntu.com/ubuntu xenial/universe amd64 checkinstall amd64 1.6.2-4ubuntu1 [121 kB]
已下载 121 kB，耗时 1秒 (61.8 kB/s)
正在选中未选择的软件包 checkinstall。
(正在读取数据库 ... 系统当前共安装有 242645 个文件和目录。)
正准备解包 .../checkinstall_1.6.2-4ubuntu1_amd64.deb ...
正在解包 checkinstall (1.6.2-4ubuntu1) ...
正在处理用于 man-db (2.7.5-1) 的触发器 ...
正在设置 checkinstall (1.6.2-4ubuntu1) ...
day@day-OptiPlex-7050:~$


day@day-OptiPlex-7050: ~
day@day-OptiPlex-7050:~$ sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
libreadline5 libssl1.0.0 libtinfo-dev
建议安装：
ncurses-doc
下列【新】软件包将被安装：
libncursesw5-dev libreadline-gplv2-dev libreadline5 libtinfo-dev
下列软件包将被升级：
libssl-dev libssl1.0.0
升级了 2 个软件包，新安装了 4 个软件包，要卸载 0 个软件包，有 557 个软件包未被升级。
需要下载 2,926 kB 的归档。
解压缩后会消耗 2,596 kB 的额外空间。
您希望继续执行吗？ [Y/n] Y
获取:1 http://cn.archive.ubuntu.com/ubuntu xenial-updates/main amd64 libssl-dev amd64 1.0.2g-1ubuntu4.15 [1,344 kB]
获取:2 http://cn.archive.ubuntu.com/ubuntu xenial-updates/main amd64 libssl1.0.0 amd64 1.0.2g-1ubuntu4.15 [1,084 kB]
获取:3 http://cn.archive.ubuntu.com/ubuntu xenial/main amd64 libtinfo-dev amd64 6.0+20160213-1ubuntu1 [77.4 kB]

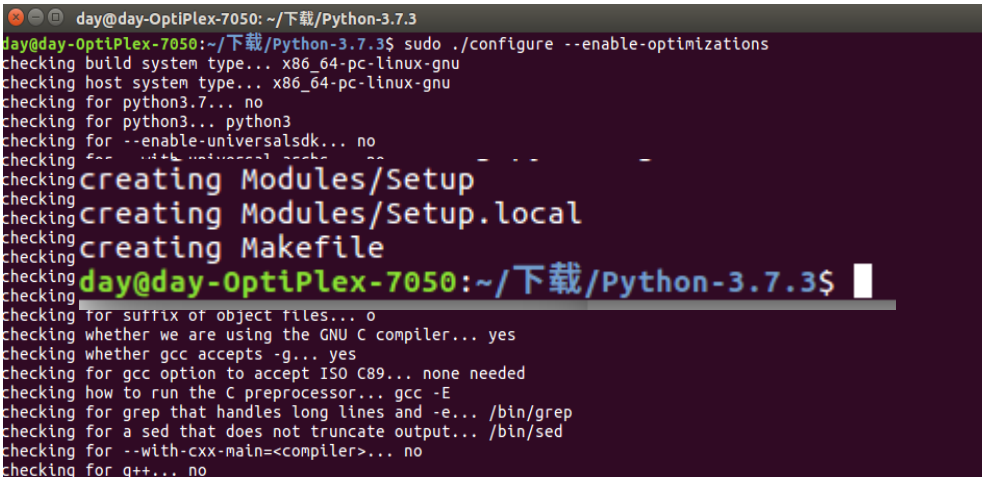
4. 在做好以上先决步骤后，使用安装命令对源代码进行编译和安装。从 python 网站下载源码，并解压到当前目录后，在终端中输入一下命令

```
cd   /home/day/下载/Python-3.7.3
sudo ./configure –enable-optimizations
sudo make altinstall
```

上边使用的 altinstall 是在系统上编译 python 源代码

make altinstall 用于防止替换默认的 python 二进制文件/ usr / bin / python

装结束后出现下图



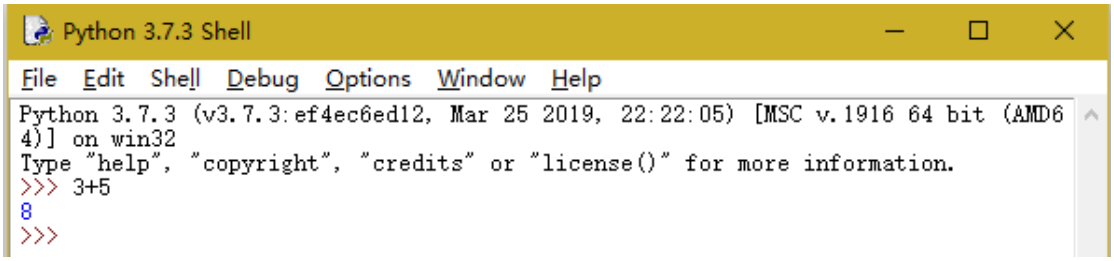
5. 编译结束后若出现出现 ModuleNotFoundError: No module named '_ctypes'，需要使用下述语句来解决问题

```
udo apt-get update
sudo apt-get upgrade
sudo apt-get dist-upgrade
sudo apt-get install build-essential python-dev python-setuptools python-pip python-smbus
sudo apt-get install build-essential libncursesw5-dev libgdbm-dev libc6-dev
sudo apt-get install zlib1g-dev libsqlite3-dev tk-dev
sudo apt-get install libssl-dev openssl
sudo apt-get install libffi-dev
```

1.3.3 Python 的集成开发环境

Python 安装完成后，本身已经集成了一个交互式开发环境（IDLE）。在 Windows10 下我们可以通过搜索 IDLE 来启动，如下图所示。IDLE 是一个 Python 的“外壳”，其实可以理解为就是一个通过输入文本与程序交互的途径。像 windows 的 cmd 窗口，像 linux 那个的命令窗口，它们都是 shell，利用它们，就可以给操作系统下达命令。同样，可以利用 IDLE 这个 shell 与 Python 进行互动。

下图就是我们熟悉的界面，这跟用命令行启动的 py 交互式解释器的显示是完全一样的。只是这个解释器中，菜单栏集成了文件编辑和调试功能。做一个简单的加法运算 3+5 回车看看效果，将马上显示结果为 8。



1.3.4 PyCharm 编译器

之前介绍的交互式开发环境（IDLE）是默认的开发环境，在 Python 开发环境中，还有很多

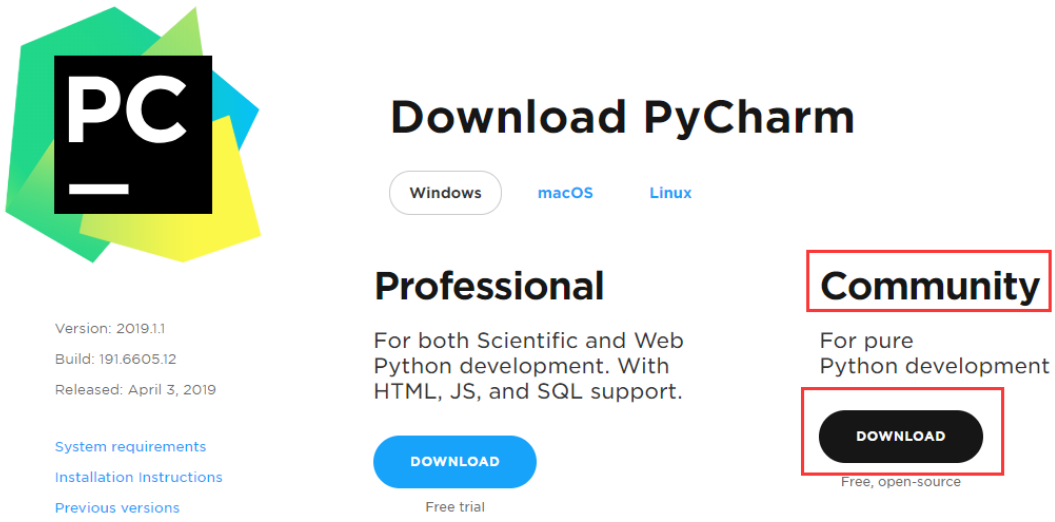
扫码看视频 1.2

第 3 方开发环境，其中 PyCharm 是整个 IDE 中综合性能最高的，堪称 IDE 中的瑞士军刀。PyCharm 是一款由 JetBrains 开发的优秀 IDE，在官网中提供了两个版本一个是专业版，一个是社区版。专业版功能比较全面强大，但社区版是开源免费的。如果有一个合法的 edu 邮箱，可以通过 JetBrains 获得学生渠道的免费专业版，但就目前入门学习开发社区版已经完全够用了。

PyCharm 具备自动补全变量和函数，提示语法错误和潜在问题，并且严格按照 PEP8 纠正编码习惯，同时也内置了交互式解释器。

1.3.5 PyCharm 的安装与初始化

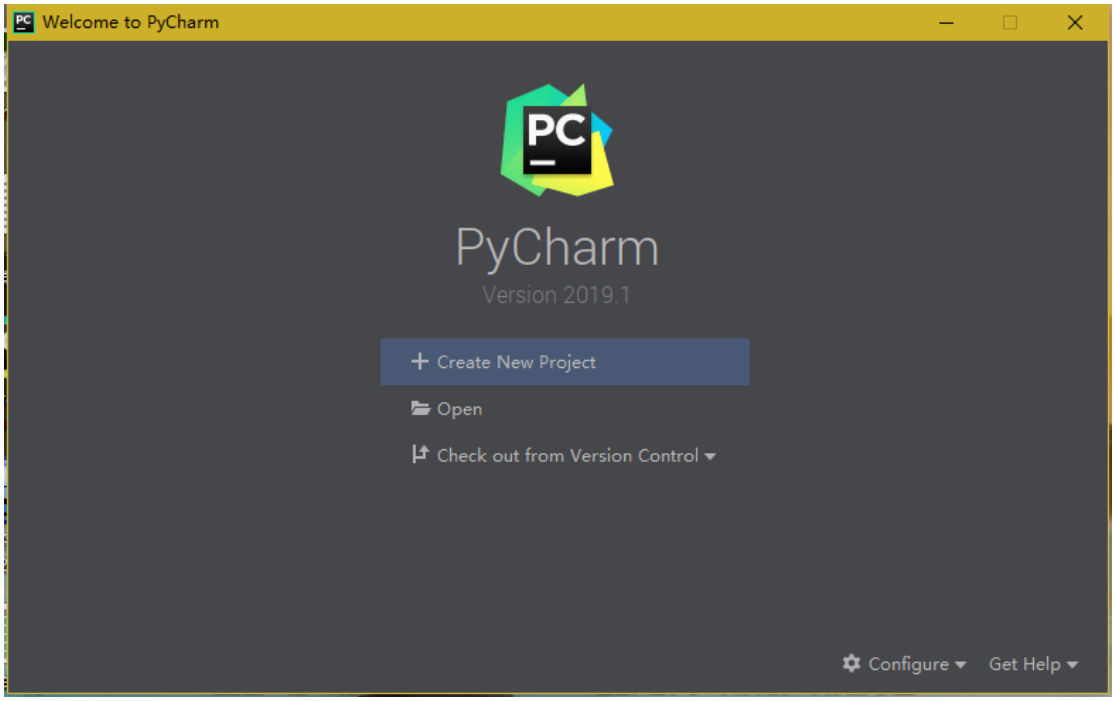
PyCharm 的官方下载地址是：http://www.jetbrains.com/pycharm/download/#section=windows。下载 PyCharm 安装包，根据自己电脑的操作系统进行选择，对于 windows 系统选择下图的所包含的安装包。



下载后，按默认值依次点击下一步。首次运行会出现下图界面

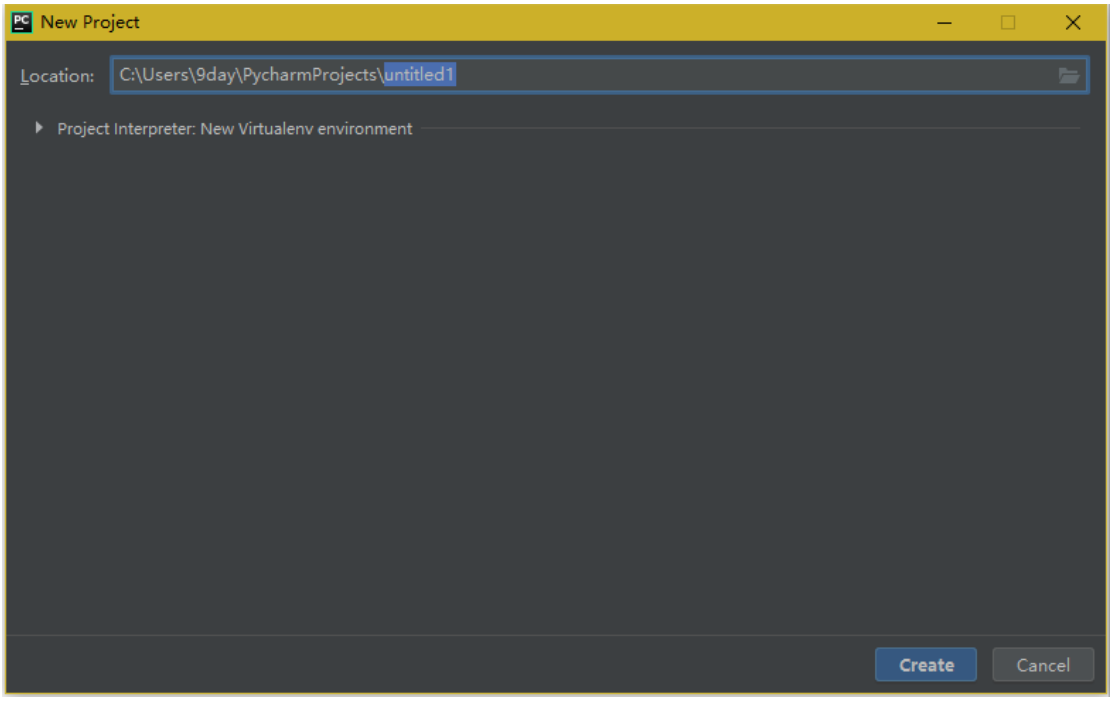
如果要下载其他历史版本可进入下列网站

https://www.jetbrains.com/pycharm/download/previous.html



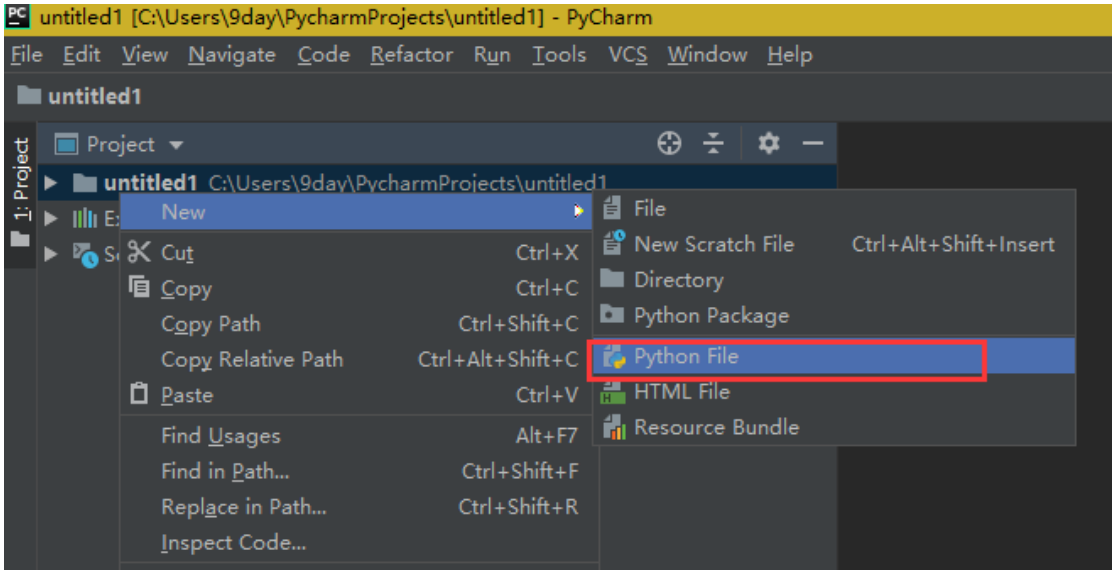
1.4 创建第一个程序

启动程序后，选择“Create New Project”，进入如下图的界面：

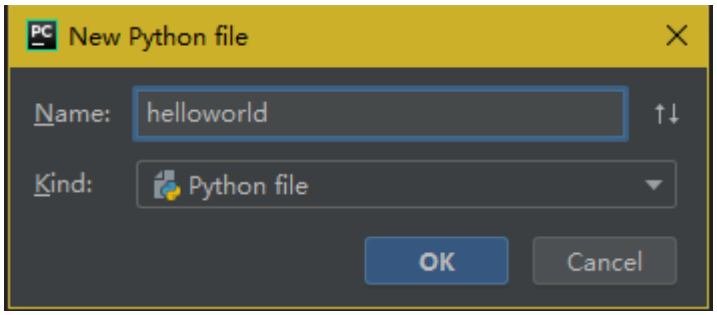


自定义项目存储路径，IDE 默认会关联 Python 解释器。选择好存储路径后，点击 create。

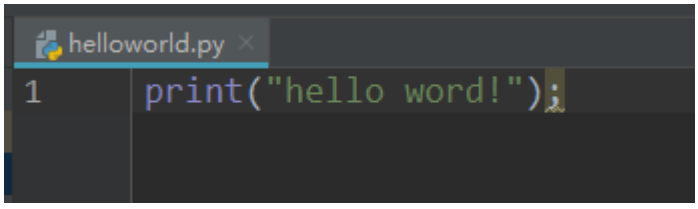
进入的界面如下图所示，鼠标右击图中箭头指向的地方，然后选择 New，最后选择 python file，在弹出的框中填写文件名（任意填写），本例填写：helloworld。

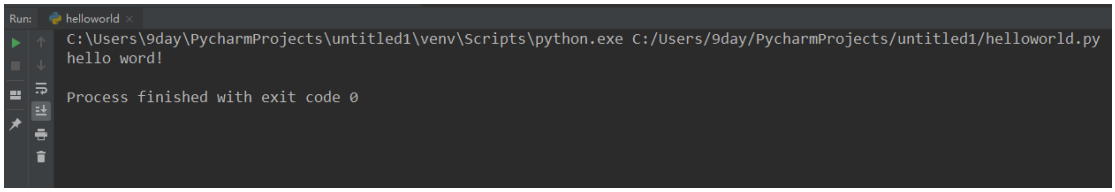


之后得到下图，然后点击 OK 即可：



文件创建成功后便进入如下的界面，编写第一个示例程序“输出 Hello World 字符串”。使用快捷键 Alt+shift+F10 运行查看效果。





之前介绍过 Python 程序的简单易用性，为了让大家更直观的体验到它的简单易用，同样是 Hello World 的程序我们看看 Java 和 C++是怎么写的，进行一个简单的对比。

Java 的 Hello World 程序代码：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

C++的 Hello World 程序代码：

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello,World!\n";
    return 0;
}
```

而 Python 只需要这样就可以了！

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

1.5 Python 的注释

程序加注释对程序设计者本身是一个标记，在大型程序中，能及时有效的进行维护/修改。对程序阅读者来说，是一个解释，能让读者通彻的了解程序和设计者的思路。对企业来说，在人员接替时能保证稳定过渡。一般 C/C++这些的注释有/* */ 和//，前面那种可以多行，从/*开始到*/之间的都将是注释。//的话仅限于该符号同行后面的内容。

Python 中的注释有单行注释和多行注释：

Python 中单行注释以 # 开头，例如：

```
# 这是一个注释
print("Hello, World!")
```

多行注释用三个单引号 ''' 或者三个双引号 """ 将注释括起来，例如:

1、单引号（'''）

```
#!/usr/bin/python3
'''
这是多行注释，用三个单引号
这是多行注释，用三个单引号
这是多行注释，用三个单引号
'''
print("Hello, World!")
```

2、双引号（"""）

```
#!/usr/bin/python3
"""
这是多行注释，用三个双引号
这是多行注释，用三个双引号
这是多行注释，用三个双引号
"""
print("Hello, World!")
```

在本书所使用的 IDE 环境 PyCharm 中，有一种快速注释方法，使用组合键“Ctrl+/”。具体操作是：选中需要注释的代码或者文字，按组合键“Ctrl+/”即可快速添加注释，这个组合键在日后学习和开发过程中将会被经常用到。

1.6 Python2.x 和 Python3.x 差异

本书所有示例代码基本遵循 Python3.x 的语法。Python 的 3.0 版本，常被称为 Python 3000，或简称 Py3k。相对于 Python 的早期版本，这是一个较大的升级。为了不带入过多的累赘，Python 3.0 在设计的时候没有考虑向下相容。许多针对早期 Python 版本设计的程式都无法在 Python 3.0 上正常执行。为了照顾现有程序，Python 2.6 作为一

个过渡版本，基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。本书建议新建 Python 程序使用 Python 3.0 版本的语法。除非执行环境无法安装 Python 3.0 或者程式本身使用了不支持 Python 3.0 的第三方库。目前不支持 Python 3.0 的第三方库有 Twisted, py2exe, PIL 等。大多数第三方库都正在努力地相容 Python 3.0 版本。即使无法立即使用 Python 3.0，也建议编写相容 Python 3.0 版本的程式，然后使用 Python 2.6, Python 2.7 来执行。

Python 3.0 的变化主要在以下几个方面：

print 函数

print 语句没有了，取而代之的是 print()函数。 Python 2.6 与 Python 2.7 部分地支持这种形式的 print 语法。在 Python 2.6 与 Python 2.7 里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #这里需要注意的是 print 后面有个空格
print("fish") #print()不能带有任何其它参数
```

然而，Python 2.6 实际已经支持新的 print()语法：

```
from __future__ import print_function
print("fish", "panda", sep=', ')
```

Unicode

Python 2 有 ASCII str() 类型，unicode() 是单独的，不是 byte 类型。现在， 在 Python 3，我们最终有了 Unicode (utf-8) 字符串，以及一个字节类：byte 和 bytearrays。

由于 Python3.X 源码文件默认使用 utf-8 编码，这就使得以下代码是合法的：

```
>>> 中国 = 'china'
>>>print(中国)
China
```

Python 2.x

```
>>> str = "我爱北京天安门"
>>> str
'\xe6\x88\x91\xe7\x88\xb1\xe5\x8c\x97\xe4\xba\xac\xe5\xa4\xa9\xe5\xae\x89\xe9\x97\xa8'
>>> str = u"我爱北京天安门"
>>> str
u'\u6211\u7231\u5317\u4eac\u5929\u5b89\u95e8'
```

Python 3.x

```
>>> str = "我爱北京天安门"
>>> str
'我爱北京天安门'
```

上述仅仅列举了两个方面的差异，考虑到初学者对面向对象编程的概念尚未接触到，诸如异常、八进制面量表示、不等运算符、很多 2.x 的模块名称在 3.x 上已被新的名字代替、数据类型等都有了变更，这些内容将不在本单元全面介绍。

1.7 小试牛刀

【例 1.1】根据圆的半径计算圆的周长和面积

编写程序，从键盘输入圆的半径，计算并输出圆的周长和面积。计算圆的周长和面积需要使用π 的值，Python 的 math 模块中包含常量 pi，通过导入 math 模块可以直接使用该值，然后使用周长和面积公式计算即可，考虑到第一次调试程序，建议直接使用 3.14 数字代入运算公式，代码如下：


扫码看视频 1.3

```
#下边的 input 是将读取键盘输入的数据存入 radius 变量中，这个变量将在后续的代码中视同为就是数学公式中的半径 r
radius=input("请输入半径")
```

```
    #下边的 2*3.14 就是一个算术运算，*字符代表数学中的乘号，其含义是 2πr²计算的结果存放到 circumference 变量中
    circumference=2*3.14*radius

    #下边这一行也同样是一个算术运算πr²    π是 3.14；r 是 radius
    area=3.14*radius*radius
    print ( "圆的周长是:    %.2f" % circumference)
    print ( "圆面积是: %.2f"% area)
```

运行结果：

```
C:\Users\9day\PycharmProjects\untitled1\venv\Scripts\python.exe C:/Users/9day/PycharmProjects/untitled1/helloworld.py
请输入半径2.0
Traceback (most recent call last):
  File "C:/Users/9day/PycharmProjects/untitled1/helloworld.py", line 3, in <module>
    area=3.14*radius*radius
TypeError: can't multiply sequence by non-int of type 'float'

Process finished with exit code 1
```

出现这个问题主要是 input 函数返回值是字符串，这里需要 float 进行类型转换，在后续单元中会单独介绍数据类型的转换，这里做一个简单的了解即可。

```
    radius=float(input("请输入半径"))
    circumference=2.0*3.14*radius
    area=3.14*radius*radius
    print ( "圆的周长是:    %.2f" % circumference)
    print ( "圆面积是: %.2f"% area)
```

运行结果：

```
    请输入半径 3
    圆的周长是:    18.84
    圆面积是: 28.26
```

【例 1.2】：摄氏温度转华氏温度

华氏温度数值上是摄氏温度的 1.8 倍多 32，所以在转换时只需要根据用户输入的摄氏温度值加上 32 即可。具体代码如下：

```
    # 输入温度
    a = float(input('请输入摄氏温度：'))

    # 转换温度
    c = a * 9 / 5 + 32
```

```
# 输出输出结果
print("摄氏温度{}转换为华氏温度为：{}".format(a, c))
```

本例输入摄氏温度 27°，计算结果如下：

```
请输入摄氏温度：27
摄氏温度 27.0 转换为华氏温度为：80.6
```

【例 1.3】用三行 print，完成以下信息的显示:

```
==================================
=    欢迎进入到身份认证系统 V1.0
= 1. 登录
= 2. 退出
= 3. 认证
= 4. 修改密码
==================================
```

上述文字是一个以字符组成的图形界面不是程序代码是运行后的效果图

程序代码如下：

```
print("="*50)    #这里表示打印 50 个=号
print("=    欢迎进入到身份认证系统 V1.0")   #这句表示在屏幕打印双引号内的字符
print("= 1.登录")
print("= 2.退出")
print("= 3.认证")
print("= 4.修改密码")
print("="*50)    #这里表示打印 50 个=号
```

【例 1.4】编写程序，从键盘获取一个个信息，然后按照下面格式显示

扫码看视频 1.4

```
==============================

姓名：张三

QQ：123456789

手机号：987654321

公司地址：北京朝阳区

==============================
```

代码如下：

```
    #下边的代码，双引号内的字符是在屏幕输出这段文字，同时将键盘输入的字符存储在 name 变量中
    name = input("请输入名字>>> ")

    #下边的代码，双引号内的字符是在屏幕输出这段文字，同时将键盘输入的字符存储在 QQ 变量中、存储在 phone_num 变量中，存储在 com_addr 变量中
    QQ = input("请输入 qq>>> ")
    phone_num = input("请输入手机号>>> ")
    com_addr = input("请输入公司地址>>> ")

    print("="*30)                          #之前介绍过，这里表示在屏幕上打印 30 个*号
    print("\n 姓名：",name)                 #\n 代表是换行，在后续的单元会单独介绍
    print("\nQQ：",QQ)
    print("\n 手机号：",phone_num)
    print("\n 公司地址：",com_addr)
    print("="*30)
```

关于上述代码中 print 函数内的一些字符特殊意义将在后续单元中详细介绍，这里我们重点是尝试通过搭建好的开发环境中进行代码的编译运行。

【例 1.5】关于 Python 的一个彩蛋

第一单元我想用 Python 的一个语句来体现 Python 的设计理念，同时也是 Python 的一个彩蛋，那就是在 IDE 或者解释器交互环境中输入 import this，将会在屏幕上输出一段 Python 之禅格言，作者是内核开发者 Tim Peters，Guido 将 Python 语言设计指导原则浓缩为了 19 条开发哲学。

```
    The Zen of Python, by Tim Peters

    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
    Complex is better than complicated.
    Flat is better than nested.
    Sparse is better than dense.
    Readability counts.
```

Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

这段英文的翻译如下：

中文版(译者 Tim Peters)
Python 之禅 by Tim Peters
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，风格相似）
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 风格的代码）
当存在多种可能，不要尝试去猜测
而是尽量找一种，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，那肯定不是一个好方案；反之亦然（方案测评标准）
命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）

关于 import this 这个彩蛋的来由以及 《The Zen of Python》 的历史故事，是发生在一次 Pycon 大会上，主办方想给大会定一个 Slogan 印在 T 恤上，然后大家奇思异想，最后几百候选名单出选出了 import this，紧接着这个彩蛋放在了 Python 2.2.1 发布。感兴趣的可以参考 https://www.wefearchange.org/2010/06/import-this-and-zen-of-python.

html。

第二单元：Python基础知识

计算机程序设计的目的是存储和处理数据，将数据分为合理的类型既可以方便数据处理，又可以提高数据的处理效率，节省存储空间。用计算机语言书写的程序称为源程序，也叫源代码。书写程序要注意语句的格式、语法约束、保留字等。

2.1 基本语法

Python 语言与 Perl，C 和 Java 等语言有许多相似之处。但是，也存在一些差异。编写 Paython 程序之前需要对语法有所了解，才能编写规范的 Python 程序。

1. Python 语句的缩进

Pyhton 语言与 Java、C#等编程语言最大的不同点是，Python 代码块使用缩进对齐表示代码逻辑，而不是使用大括号。这对习惯用大括号表示代码块的程序员来说，确实是学习 Python 的一个障碍。

Python 每段代码块缩进的空白数量可以任意，但要确保同段代码块语句必须包含相同的缩进空白数量。

【例 2.1】由于缩进没有对齐而产生的语法错误

```
#IF 语句示例
a=input("请输入第一个数")
b=input("请输入第二个数")
if a > b:
        print('a>b')
else:
print('a<b')
```

在上述代码中，if 语句后缩进的 1 行构成一个代码块，else 语句

后缩进的 1 行也构成一个代码块。如果同一代码块中各语句前的空格数不一致（else 语句的 print 函数和 if 语句的 print 函数没有缩进对齐，产生语法错误。），运行时将会报告出错信息，如下所示。

```
  File "D:/工作/python/案例源代码/第二单元/ex2.1.py", line 7
    print('a<b')
        ^
IndentationError: expected an indented block
```

代码应更正为：

```
#IF 语句示例
a=input("请输入第一个数")
b=input("请输入第二个数")
if a > b:
      print('a>b')
else:
      print('a<b')
```

运行结果：

```
请输入第一个数 6
请输入第二个数 8
a<b
```

建议在代码块的每个缩进层次使用单个制表符或两个空格或四个空格 ，切记不能混用。不同文本编辑器中的制表符（Tab 键）表示的空白宽度不一致，如果使用的代码要跨平台使用，建议不用使用制表符。

2. Python 的多行语句

Python 语句一般以新的一行作为前面语句的结束。但在一些情况下，有可能一条语句需要在多行输出，如语句过长，导致编辑器的窗口宽度不能完全显示时。可以在语句的外部加上一堆圆括号来实现，也可以使用“\”（反斜杠）来实现分行书写功能。

与写在圆括号中的语句类似，写在[]、{}内的跨行语句被视为一

行语句，不再需要使用圆括号换行。

【例 2.2】Python 语句的分行书写

```
strl1 = ("一句语句过长，导致编辑器的窗口宽度不能完全显示时。可以\
在语句的外部加上一堆圆括号来实现，也可以使用反斜杠来实现\
分行书写功能。")
Strl2 = ("一句语句过长，导致编辑器的窗口宽度不能完全显示时。可以"
         "在语句的外部加上一堆圆括号来实现，也可以使用反斜杠来实现"
         "分行书写功能。")
week =['Monday','Tuesday','Wednesday','Thursday','Friday',' Saturday','Sunday']
print(strl1 )
print(Strl2)
print(week)
```

运行结果：

```
一句语句过长，导致编辑器的窗口宽度不能完全显示时。可以在语句的外部加上一堆圆括号来实现，也可以使用反斜杠来实现分行书写功能。
一句语句过长，导致编辑器的窗口宽度不能完全显示时。可以在语句的外部加上一堆圆括号来实现，也可以使用反斜杠来实现分行书写功能。
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', ' Saturday', 'Sunday']
```

3. Python 引号

在 Python 语言中，引号主要用于表示字符串。可以使用单引号（'）、双引号（"）、三引号（'''），引号必须成对使用。单引号和双引号用于程序中的字符串表示；三引号允许一个字符串可以跨多行、字符串中可以包含换行符、制表符以及其他特殊字符，三引号也用于程序中的注释。

2.2 变量及其基本数据类型

2.2.1 变量


扫码看视频 2.1

用标识符命名的存储单元的地址称为变量，也叫内存变量。变量是用来存储数据的，通过标

识符可以获取变量的值，也可以对变量进行赋值。对变量赋值的意思是将值赋给变量，赋值完成后，变量所指向的存储单元存储了被赋的值，在 Pyhton 语言中赋值操作符为“=、+=、-=、*=、/=、%=、**=、//=”。

变量是内存中命名的存储位置，其值可以动态变化。Python 中的变量不需要声明，可以直接使用赋值运算符对其进行赋值运算，并根据所赋的值决定其数据类型。

```
result = 30;                #定义一个整型变量
name=”Peter”                #定义一个字符串变量
```

关于变量，使用时注意以下方面：

一是计算机语言中的赋值是一个重要的概念。若 x=2，赋值运算的含义是将 2 赋予变量 x，若 x=x+1，赋值运算的含义是将 x 加 1 之后的值再赋予 x，x 的值是 3，这与数学中的等于含义是不同。

二是 Python 中的变量具有类型的概念，变量的类型由所赋的值来决定。在 Python 中，只要定义了一个变量，并且变量存储了数据，那么变量的数据类型就已经确定了，系统会自动识别变量的数据类型。例如，若 x=2，则 x 是整型数据；若 x=‘hello’，则 x 是一个字符串类型。

2.2.2 基本数据类型

计算机程序设计的目的是存储和处理数据，将数据分为合理的类型既可以方便数据处理，又可以提高数据的处理效率，节省存储空间。Python 的数据类型指明了数据的状态和行为，包括数值类型（Number）、字符串类型（Str）、列表类型（List）、元组类型等。其

中，数值类型是 Python 的基本数据类型，包含整型（int）、浮点型（float）、复数类型（complex）和布尔类型（bool）4 种。

程序使用变量来临时保存数据，变量使用标识符来命名。

1. 整数类型

整数类型简称整型，它与数学中整数的概念一致。在 Python 中整数类型被指定为 int 类型。整数类型对应于数学中的整数概念。可以执行的算法有+、-、*、/ 以及一些其他操作。默认情况下，整数采用的是十进制，但在方便的时候也可以使用其他进制，分别是二进制(以“0B”或“0b”开头)、八进制（以数字“0O”或”0o”开头）和十六进制（以“0X”或“0x”开头)）。

Python 的整型数据理论上的取值范围是（-∞，∞），实际的取值范围受限于运行 Python 程序的计算机内存大小。下面是一些整型类型的数据：

```
98, 19, 0B 1101，0b 1011，0O157，0o235，0X1AB, 0x259E
```

【例 2.3】整型类型测试

```
#整型类型测试
a=0o104
b=0B1011
c=0x2BEF
print(a,b,c)
print(type(a),type(b),type(c),)
```

运行结果：

```
68 11 11247
<class 'int'> <class 'int'> <class 'int'>
```

2. 浮点型

浮点型用于表示数学中的实数，是带有小数的数据类型。例如，3.14、11.6 都属于浮点型。浮点型可以用十进制或科学计数法表示。

下面是用科学计数法表示的浮点型数据：

```
3.56e2, 0.25e6，1.5e-3
```

E 或 e 表示基数是 10，后面的整数表示指数，指数的正负使用+号或-号表示，其中，+可以省略。需要注意的是，Python 的浮点型占用 8 个字节，能表示的数的范围是-1.8^{308} ~1.8^{308}。

3. 复数类型

复数类型用于表示数学中的复数，一般形式为 x＋yj。其中的 x 是复数的实数部分，y 是复数的虚数部分，这里的 x 和 y 都是实数。例如，5+3j、-3.4-6.8j 都是复数类型。多数计算机语言设有复数类型，一个复数必须有表示虚部的实数和 j，如 1j、-1j 都是复数，而 0.0 不是复数，并且表示虚部的实数部分即使是 1 也不可以省略。

【例 2.4】复数类型测试

```
#复数类型测试
f1=4.5+3j
print(f1)
print(type(f1))
print(f1.real)
print(f1.imag)
```

运行结果：

```
(4.5+3j)
<class 'complex'>
4.5
3.0
```

4. 布尔类型

布尔类型可以看作是一种特殊的整型，所有内置的数据类型与标准库提供的数据类型都可以转换为一个布尔型值。Python 提供了 3 个逻辑操作符：and、or、not。

布尔型数据只有两个取值：True 和 False。如果将布尔值进行数

值计算，True 会被当做整型 1，False 会被当作整型 0。每一个 Python 对象都自动具有布尔值，进而可用于布尔测试。以下对象的布尔值都是 False，包括 none、false、整型 0、浮点型 0.0、复数 0.0+0.0j、空字符串“ ” 、空列表[ ]、空元组（）、空字典{ }，这些数据的值可以用 Python 的内置函数 bool（）来测试。

【例 2.5】布尔类型测试

```
#测试布尔类型
a1=0
print(type(a1),bool(a1))
a2=0.0
print(type(a2),bool(a2))
a3=0.0+0.0j
print(type(a3),bool(a3))
a4=""
print(type(a4),bool(a4))
a5=[]      #列表类型
print(type(a5),bool(a5))
a6={}      #字典类型
print(type(a6),bool(a6))
```

运行结果：

```
<class 'int'> False
<class 'float'> False
<class 'complex'> False
<class 'str'> False
<class 'list'> False
<class 'dict'> False
```

5. 字符串类型

Python 的字符串是用单引号、双引号和三引号括起来的字符序列，用于描述信息。如'Python is wonderful!'、'1929288338'、'张三'、''等。其中，''表示空字符串。字符串和数字一样，都是不可变对象。所谓不可变，是指不能原地修改对象的内容。字符串的运算和操作将在第三章介绍。

2.3 Python 数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。Python Number 类型转换如下：

函数名	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 转换到一个浮点数
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)	用来计算在字符串中的有效 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
chr(x)	将一个整数转换为一个字符
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)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

表 2-1 Python Number 类型转换

2.4 标识符和关键字

扫码看视频 2.2

标识符和关键字是计算机语言的基本语法元素，是编写程序的基础，不同计算机语言的标识符和关键字略有区别。

2.4.1 标识符

计算机中的数据，如一个变量、方法、对象等都需要有名称，以方便程序调用。这些用户定义的、由程序使用的符号就是标识符。标识符是计算机语言中允许作为名字的有效字符串集合。Python 标识

符字符串规则和其他大部分用 C 编写的高级语言相似，用户可以根据程序设计的需要来定义标识符，标识符的命名需要遵循下面的规则。

●标识符可以由字母（大写 A—Z 或小写 a—z）、数字（0—9）和_（下划线）组合而成，但必须以字母或者下划线开始。数字不能作为首字符。当名字包含多个单词时，可以使用下划线_来连接，例如 monty_Python。

●标识符不能包含除_以外的任何特殊字符，如：%、#、&、逗号、空格等。

●标识符不能包含空白字符（换行符、空格和制表符称为空白字符）。

● 标识符不能是 Python 语言的关键字和保留字；

● 标识符区分大小写，num1 和 Num1 是两个不同的标识符。

●标识符的命名尽量符合见名知义原则，从而提高代码的可读性，例如，名字 就定义为 name，定义学生用 student。

正确标识符的命名示例

width、height、book、result、num、num1、num2、book_price

错误标识符的命名示例

123rate（以数字开头）、Book Author（包含空格）、Address#（包含特殊字符）、class（class 是类关键字）

Python 标识符还遵循以下一些约定。

第一条约定：不要使用 Python 预定义的标识符名对自定义的标识符进行命名。Python 内置数据类型名（比如 int、float、list、str 与 tuple）应避免被使用，Python 内置函数名与异常名作为标识符名也应避免被使用。如何判断自己对标识符的命名是否正确。Python 有一

个内置的名为 dir ()的函数，该函数可以返回对象属性列表。

第二条约定：应避免名称开头和结尾都使用下划线（_）。开头和结尾都使用下划线表示的名称表示 Python 自定义的特殊方法与变量。对于特殊方法，可以对其进行重新实现，也就是给出实现版本，但不应该再引入这种开头和结尾都使用下划线的名称。

2.4.2 关键字

Python 预先定义了一部分有特别意义的标识符，用于语言自身使用。这部分标识符称为关键字或保留字，不能用于其它用途，否则会引起语法错误，随着 Python 语言的发展，其预留的关键字也会有所变化。Python 常用的关键字如表所示。

and	del	from	not	while	as	elif
global	or	with	assert	else	if	pass
yield	break	except	import	print	class	exec
in	raise	continue	finally	is	return	def
for	lambda	try				

表 2-2 Python 常用的关键字

【例 2.6】输出 Python 关键字

```
# python 中的 33 个 key(基本语法):
#  代码命令如下
from keyword import kwlist      # kwlist 列表中包含了所有的关键字
print(kwlist)
```

运行结果：

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

2.5 Python 的运算符

扫码看视频 2.3

所谓运算符，指的是运算符号，用于将各种类型的数据进行运算，让静态的数据跑起来。运算符是程序设计语言最基本的元素，也是构成表达式的基础。在 Python 中，有的时候需要对一个或多个数字，一个或多个字符串，以及其他的数据对象进行运算操作，此时需要用到运算符。比如 2+3 中的“+”是一种运算符。每种运算中所包含的符号称为相应的运算符，如算术运算符、比较运算符等。

编程语言中的运算大致分为以下几个大类：

- 算术运算，用于加减乘除等数学运算
- 赋值运算，用于接收运算符或方法调用返回的结果
- 比较运算，用于做大小或等值比较运算
- 逻辑运算，用于做 与、或、非运算
- 位运算， 用于二进制运算

2.5.1 算术运算

算术运算可以完成数学中的加减乘除四则运算。

运算符	说明	实例
+	两个对象相加	2 + 3 结果为 5
-	两个对象相减	3 - 2 结果为 1
*	两个数相乘或返回一个重复若干次的序列	2 * 3 结果为 6; 'abc' * 2 结果为 'abcabc'
/	两个数相除	3 / 2 结果为 1.5
//	整除，返回商的整数部分	3 // 2 结果为 1，3 // 2.0 结果为 1.0
%	求余/取模，返回除法的余数	3 % 2 结果为 1，3 % 2.0 结果为 1.0
**	求幂/次方	2 ** 3 结果为 8

表 2-3　Python 的算术运算

【例 2.7】两个数字相加

```
#两个数字相加
a=7+8
print(a)
```

运算结果：

```
15
```

2.5.2 赋值运算

赋值运算符用于计算表达式的值并送给变量。在 Python 中，赋值运算有以下 3 种情况：为单一变量赋值；为多个变量赋一个值；为多个变量赋多个值。赋值运算是将赋值号右边的值送给赋值号左边的变量，赋值表达式的运算方向是从右到左。例如 a=a+1 就是一个合法的赋值运算，先计算 a+1 的值，再送给赋值号左边 a，这和数学中的等式是完全不同的含义。表 2-4 列举了 Python 中的赋值运算符，其中 a= 5, b= 3。

运算符	描述	实例
=	简单赋值运算符	c = a − b, c 计算后的结果是 2
+=	加法赋值运算符	a += b 相当于 a = a + b, a 计算后的结果是 8
-=	减法赋值运算符	a -= b 相当于 a = a −b, a 计算后的结果是 2
*=	乘法赋值运算符	a *= *b 相当于 a = a*b, a 计算后的结果是 15
/=	除法赋值运算符	a /= b 相当于 a = a / b, a 计算后的结果是 1.6666667
//=	取整除赋值运算符	a //= b 相当于 a = a // b, a 计算后的结果是 1
%=	取余赋值运算符	a %= b 相当于 a = a % b, a 计算后的结果是 2
**=	幂赋值运算符符	a **= b 相当于 a = a**b, a 计算后的结果是 125

表 2-4　Python 的赋值运算

【例 2.8】对数进行冥计算

```
#"**": 求冥计算
a=5
b=3
a **= b
print(a)
```

运算结果：

```
125
```

2.5.3 比较运算

比较运算是指两个数据之间的比较运算。比较运算符多用于数值型数据的比较，有时也用于字符串数据的比较，比较的结果是布尔值 True 或 False。用比较运算符连接的表达式称为关系表达式，一般在程序分支结构中使用。

Python 有 8 种比较操作，它们具有相同的优先级。比较操作可以被任意连接，比如 x< y <= z 等同于 x < y and y <= z，只是第一种形式下的 y 只被评估一次。另外，当 x < y 不成立时，这两种形式下的 z 都不会被评估。

运算符	说明	实例
<	严格小于	3 < 5 结果为 True，5 < 5 结果为 False
<=	小于或等于	3 <= 5 结果为 True，5 <= 5 结果为 True
>	严格大于	5 > 3 结果为 True，5 > 5 结果为 False
>=	大于或等于	5 >= 3 结果为 True，5 >= 5 结果为 True
==	等于	“hello”==“hello”结果为 True
!=	不等于	3!=5 结果为 True
is	判断两个标识符是否引用自一个对象	x is y，如果 id(x) == id(y)，即 x 也 y 的指向同一个内存地址，则结果为 1，否则结果为 0
is not	判断两个标识符是否引用自不同对象	x is not y，如果 id(x) != id(y)，即 x 和 y 指向不同的内存地址，则结果为 1，否则结果为 0

表 2-5　Python 的比较运算

【例 2.9】等于运算符

```
#"==：比较两个对象是否相等
a=5==6
print(a)
b ="yes"=="yes"
print(b)
```

运算结果：

```
False
True
```

2.5.4 逻辑运算

逻辑运算符包括 and、or、not，分别表示逻辑与、逻辑或、逻辑

非，运算的结果是布尔值 True 或 False。下面按照他们的优先级升序顺序进行说明：其中 x=5、y=0

运算符	说明	实例
or	如果 x 是 True,它返回 x 的值,否则它返回 y 的计算值。	x or y,值为 5
and	如果 x 为 False, x and y 返回 False,否则它返回 y 的计算值。	x and y,值为 0
not	如果 x 为 True,返回 False 。如果 x 为 False,它返回 True。	not x,值为 False not y,值为 True

表 2-6　Python 的逻辑运算

【例 2.10】逻辑非运算符

```
#逻辑非 not
x=True
y=not x
print(y)
z=False
print(not z)
```

运算结果：

```
False
True
```

说明：

- or 是一个短路操作符，也就是说，只有第一个参数的评估结果为 False 时，第二个参数才会被评估；
- and 也是一个短路操作符，也就是说，只有第一个参数的评估结果为 True 时，第二个参数才会被评估；
- not 操作符比非布尔操作符优先级低，因此，not a == b 被解释为 not (a == b)；如果写成 a == not b 会包语法错误。

2.5.5 位运算符

位运算符用于对整数中的位进行测试、置位或移位处理，对数据进行按位操作。按位运算是指把数字转换为二进制来进行计算，位运算符包括以下几种：

其中假设：

a = 60, 对应的二进制格式为 0011 1100

b = 13, 对应的二进制格式为 0000 1101

运算符	说明	实例
&	按位与：参与运算的两个值，如果相应的二进制位都为 1，则该位结果为 1，否则为 0	a & b 对应的二进制结果为 0000 1100，十进制为 12
|	按位或：参与运算的两个值，只要对应的二进制位由一个为 1 时，该位结果就为 1	a | b 对应的二进制结果为 0011 1101，十进制为 61
^	按位异或：参与运算的两个值，当对应的二进制位不同时，该为结果为 1，否则改为结果为 0	a ^ b 对应的二进制结果为 0011 0001, 十进制为 49
~	按位取反：这个是单目运算符，只有一个值参与运算，运算过程是对每个二进制位取反，即把 1 变 0，把 0 变 1	~a 的二进制结果为 1100 0011，十进制数为-61
<<	左移运算符：运算数的各二进制位全部左移若干位，高位丢弃，低位补 0，结果相当于运算数乘以 2 的 n 次方，正负符号不发生改变	a << 2 的二进制结果为 1111 0000，十进制数为 240
>>	右移运算符：运算数的各二进制位全部右移若干位，结果相当于运算数除以 2 的 n 次方，正负符号不发生改变	a >> 2 的二进制结果为 0000 1111，十进制为 15

表 2-7　Python 的位运算符

【例 2.11】#按位与运算&

```
#按位与运算&
x=7&18
print (x)
```

运算结果：

```
2
```

2.6 运算符的优先级

在一个表达式中可能包含多个有不同运算

扫码看视频 2.4

符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。表达式的结合次序取决于表达式中各种运算符的优先级。优先级高的运算符先结合，优先级低的运算符后结合，同一行中的运算符的优先级相同。Python 运算符优先级从高到低排列如表 2-8 所示。

运算符	描述
**	指数（最高优先级）
~、+、-	按位翻转，一元加号和减号（最后两个的方法名为+@和-@）
*、/、%、//	乘、除、取模和取整除
+、-	加法、减法
>>、<<	右移、左移运算符
&	位与
^、|	位运算符
<=、<、>、>=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
not	逻辑非
and	逻辑与
or	逻辑或

表 2-8　Python 的运算符优先级从高到低排列

【例 2.12】#运算符的优先级

```
#运算符的优先级
a=2*3+5<=5+1*2
#根据优先级关系，先进行乘法运算，然后进行加法运算，接着进行比较运算
print(a)
```

运算结果：

```
False
```

2.7 实验

【例 2.13】位运算

a 为 46，b 为 20，进行位运算，其代码如下。可以从打印结果直观的看到各个运算符的作用。

在实例中，a 、b 对应的二进制数分别如下：

a= 0010 1110;

b= 0001 0100;

```
#位运算
a=46
b=20
print('a&b=',a&b)
print('a|b=',a|b)
print('a^b=',a^b)
print('~a=',~a)
print('a<<2=',a<<2)
print('a>>2=',a>>2)
```

运行结果：

```
a&b= 4
a|b= 62
a^b= 58
~a= -47
a<<2= 184
a>>2= 11
```

【例 2.14】各进制间的转换

二、八、十、十六 进制间的转换，int(string_num, n)，其中 string_num 表示某种进制的字符串，n 各表示 string_num 是什么进制数。

二、八、十六 进制转为十进制：使用 int()或者 eval()。

十进制转为二、八、十六进制：使用 bin()、oct()、hex() 或者使用 format()。

b：二进制，o：八进制，d：十进制，x：十六进制

bin()、oct()、hex()返回值均为字符串，且分别带有 0b、0o、0x 前缀。

hex 函数比 format 函数慢，eval 函数比 int 函数慢。

```
# 二进制转十进制
print('二进制 1111011 转十进制',int("1111011", 2))
print('二进制 1111011 转十进制',eval("0b1111011"))

# 十进制转二进制
print('十进制 18 转二进制',bin(18))
print('十进制 18 转二进制',"{0:b}".format(18))

# 八进制转十进制
print('八进制 011 转十进制',int("011", 8))
print('八进制 011 转十进制',eval("0o011"))

# 十进制转八进制
print('十进制 011 转八进制',oct(30))
print('十进制 011 转八进制',"{0:o}".format(30))

# 十六进制转十进制
print('十六进制 12 转十进制',int("12", 16))
print('十六进制 12 转十进制',eval("0x12"))

# 十进制转十六进制
print('十进制 87 转十六进制',hex(87))
print('十进制 87 转十六进制',"{0:x}".format(87))
```

运行结果：

```
二进制 1111011 转十进制 123
二进制 1111011 转十进制 123
十进制 18 转二进制 0b10010
十进制 18 转二进制 10010
八进制 011 转十进制 9
八进制 011 转十进制 9
十进制 011 转八进制 0o36
十进制 011 转八进制 36
十六进制 12 转十进制 18
十六进制 12 转十进制 18
十进制 87 转十六进制 0x57
十进制 87 转十六进制 57
```

【例 2.15】根据输入的三科成绩值，计算平均分和总分

```
#计算平均分和总分
name=input("学生姓名：")
Chinese=float(input("语文成绩:"))
Maths=float(input("数学成绩:"))
English=float(input("英语成绩:"))
sum=Chinese+Maths+English
average=sum/3
print("总成绩=%.2f" %sum)
print("平均成绩=%.2f" %average)
```

运行结果：

```
学生姓名：lucy
语文成绩:98.5
数学成绩:95.5
英语成绩:92
总成绩=286.00
平均成绩=95.33
```

【例 2.16】已知三角形的边长求三角形的面积和周长

```
#已知三角形的边长求三角形的面积和周长
import math
a=float(input('a='))
b=float(input('b='))
c=float(input('c='))
if a+b>c and a+c>b and b+c>a:
    d=a+b+c
    e=(a+b+c)/2                    #计算半周长
    f=math.sqrt(e*(e-a)*(e-b)*(e-c))          #计算三角形面积
# sqrt() 方法返回数字 x 的平方根, sqrt()是不能直接访问的，需要导入 math 模块，通过静态对象调用该方法。
    print('三角形的周长为：'+str(d))
    print('三角形的面积为：%.2f' % f)
else:
    print('三条变得长度不能构成三角形')
```

运行结果：

```
a=3
b=5
c=6
三角形的周长为：14.0
三角形的面积为：7.48
```

第三单元：Python字符串输入输出

字符串是一种表示文本的数据类型。字符串的表示、解析和处理

是 Python 的重要内容，也是 Python 编程的基础之一。

3.1 字符串表示


扫码看视频 3.1

字符串是一种非常常见的 Python 自带的数据类型，在 Python 中用引号引起来的字符集称为字符串，比如，’ hello’、”你好吗？”、”my mother”、”5+6”等都属于字符串

Pyhton 中的字符串被定义为一个字符集合，它被引号所包围，引号可以是单引号、双引号或者三引号。其中单引号和双引号包围的是单行字符串，二者的作用相同。

使用单引号(‘)：可以用单引号指示字符串，就如同’Quote me on this‘这样。所有的空白，即空格和制表符都照原样保留。

使用双引号(“)：在双引号中的字符串与单引号中的字符串的使用完全相同，例如”What‘s your name?“。

三引号可以包围多行字符串。这种字符串常常出现在函数声明的下一行，用来注释函数的功能。三引号可以保留所有字符串的格式信息，如果字符串跨越多行，行与行之间的回车符也可以保存下来，引号、制表符或者其他任何信息都可以保存下来。利用这种方式，可以将整个段落作为单个字符保存下来进行处理。

【例 3.1】单引号

```
#单引号
c1='It is a "bird"!'
print(c1)
```

运行结果

```
It is a “bird”
```

在上述代码中，假如希望定义一个字符串变量，值为“It is a “bird”!”，此时不能使用双引号定义。因为该字符串中就含有双引号，如果外层再用双引号包含，则会出现冲突，所以如果要定义该字符串变量，可以通过单引号去定义。

【例 3.2】双引号

```
#双引号
c2="It is a bird!"
print(c2)
```

运行结果：

```
It is a bird!
```

【例 3.3】三引号

```
#三引号
c3="""dog
tiger
monkey
bird"""
print(c3)
```

运行结果：

```
dog
tiger
monkey
bird
```

3.2 转义字符


扫码看视频 3.2

转义字符用于表示一些在某些场合不能直接输入的特殊字符。代码中需要输入退格符、换行符、换页符等不可见字符，解决这个问题需要使用转义符。转义符由反斜杠（\）引导，与后面相邻的字符组成了新的含义。常用的转义符如表 3-1 所示

转义字符	描述
\(在行尾时)	续行符

****	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行
\other	其它的字符以普通格式输出

表 3-1 常用的转义符

【例 3.4】输出换行符

```
#输出换行符
print("离离原上草，\n 一岁一枯荣。")
```

运行结果：

```
离离原上草，
一岁一枯荣。
```

【例 3.5】输出双引号

如果想要输入带有引号的字符串，直接编写这段字符串就不会得到想要的结果，比如显示：He said,:"Let’s go and play footbool"。如果直接编写，则是 print("He said,:" Let’s go and play footbool "”），就会显示语句错误，Python 判断不出双引号是字符串里面的，

正确的代码是：

```
#输出双引号
print("He said,:\" Let’s go and play footbool \"")
```

运行结果：

```
He said,:" Let’s go and play footbool "
```

3.3 格式化字符串

扫码看视频 3.3

程序运行输出的结果很多时候是以字符串的形式呈现，为了实现输出的灵活性和可编辑性，需要控制字符串的输出格式，即字符串类型的格式化。Python 支持两种字符串的格式化方法，一是使用格式化操作符”%”；另一种采用专门的 str.format()方法。

3.3.1 用%操作符格式化字符串

Python 的%操作符可用于格式化字符串，控制字符串的呈现格式。格式字符串时，Python 使用一个字符串作为模板。模板中有格式符，这些格式符为显示值预留位置，并说明显示值应该呈现的格式。字符串模板的参数如表 3-2 所示。

符号	描述
*	定义宽度或者小数点精度
-	表示左对齐，正数前无符号，负数前添加负号
+	表示右对齐，正数前添加正号，负数前添加负号
<sp>	表示右对齐，正数前添加空格，负数前添加负号
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'
0	表示右对齐，显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
(var)	映射变量(字典参数)
m.n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

表 3-2 字符串模板的参数

格式控制符用于控制字符串模板中不同符号的显示，例如，可以显示为字符串、整数、浮点数等形式。字符串格式化控制符如表 3-3 所示。

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	格式化无符号整型
%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，可指定小数点后的精度
%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

表 3-3 python 字符串格式化控制符

格式化字符串时，Python 使用一个字符串作为模板。模板中有格式符，这些格式符为真实值预留位置，并说明真实数值应该呈现的格式。Python 用一个 tuple 将多个值传递给模板，每个值对应一个格式符。例如：print("I'm %s. I'm %d year old" % ('Lucy', 20))

"I'm %s. I'm %d year old" 为我们的模板。%s 为第一个格式符，表示一个字符串。%d 为第二个格式符，表示一个整数。('Lucy', 20)的两个元素' Lucy '和 20 为替换%s 和%d 的真实值。

在模板和 tuple 之间，有一个%号分隔，它代表了格式化操作。

整个"I'm %s. I'm %d year old" % ('Vamei', 99) 实际上构成一个字符串表达式。我们可以像一个正常的字符串那样，将它赋值给某个变量。具体代码如下：

【例 3.6】格式化字符串

```
#输出姓名年龄
a = "I'm %s. I'm %d year old" % ('Lucy', 20)
print(a)
```

运行结果：

```
I'm Lucy. I'm 20 year old
```

【例 3.7】输出书本数量

```
#输出书本数量
bookcount = 35
print("there are %d books"%bookcount)
```

运行结果：

```
there are 35 books
```

3.3.2 format()方法

从 Python2.6 开始，就增加了一种格式字符串的 str. format()方法，这种方法方便了用户对字符串进行格式处理。

模板字符串与 format()中参数的对应关系

str. format()方法中的 str 被称为模板字符串，其中包括多个由“{}”表示的占位符，这些占位符接收 format()方法中的参数。str 模板字符串与 format()方法中的参数对应关系有以下情况。

- 位置参数匹配

在模板字符串中，如果占位符{}为空，将会按照参数出现的先后次序进行匹配。如果占位符{}指定了参数的序号，则会按照序号替换对应参数。

【例 3.8】位置参数

```
#位置参数
print("{} is {} years old".format("Lucy",20))
print("{0} is {1} years old".format("Lucy",20))
print("hello,{0}!{0} is {1} years old".format("Lucy",20))
```

运行结果：

```
Lucy is 20 years old
Lucy is 20 years old
hello,Lucy!Lucy is 20 years old
```

- 使用键值对的关键字参数匹配

format()方法中的参数用键值对形式表示时，在模板字符串中“键”来表示。

【例 3.9】关键字参数

```
#关键字参数
print("{name} was born in {year},she is {age} years old"\
```

```
        .format(name="Lucy",age="20",year="1999"))
```

运行结果：

```
Lucy was born in 1999,she is 20 years old
```

- 使用序列的索引作为参数匹配

如果 format()方法中的参数是列表或元组，可以用其索引来匹配。

【例 3.10】下标参数

```
#下标参数
student=["Lucy",20]
school=("Yunnan","YNJD")
print("{1[0]} was born in {0[0]},she is {1[1]} years old".format(school,student))
```

运行结果：

```
Lucy was born in Yunnan,she is 20 years old
```

1.模板字符串 str 的格式控制

下面详细说明模板字符串 str 的格式控制，其语法格式如下：

[[fill]align][sign][width][,][.precision][type]

模板字符串参数的含义如下：

Fill：可选参数，空白处填充的字符。

align：可选参数，用于控制对齐方式，配合 width 参数使用，

- <：内容左对齐，
- >：内容右对齐，
- ^：内容居中对齐。

Sign：可选参数，数字前的符号。

- +：在正数数值前添加正号，在负数数值前添加负号。
- -：在正号不变，在负数数值前添加负号。
- 空格：在正数数值前添加空格，在负数数值前添加负号。
- Width：可选参数，指定格式化后的字符串所在的宽度。

- 逗号（,）：可选参数，为数字添加千分位分隔符。
- Precision：可选参数，指定小数位的精度。
- type：可选参数，指定格式化的类型。

【例 3.11】格式化字符串

```
#格式化字符串
print('{:*>10}'.format('3.14') )  #宽度 10 位，右对齐
print('{:*<10}'.format('3.14') )  #宽度 10 位，左对齐
print('{0:^10},{0:*^10}'.format('3.14'))  #宽度 10 位，居中对齐
```

运行结果：

```
******3.14
3.14******
   3.14   ,***3.14***
```

3.4 字符串的比较


扫码看视频 3.4

1．单字符字符串的比较

比较两个单字符字符串是否相同，使用“==”运算符，如果两个字符是相同的，则该表达式返回真，若不同，则返回假。进行比较时，Python 使用的是字符串的内存字节表示，此时的排序是基于 Unicode 字元的，函数 ord 和 chr 可以帮助查找字符与字符对应的 ASCII 码表中整数的关系，其中 ord()函数是 chr()函数（对于 8 位的 ASCII 字符串）的配对函数，它以一个字符作为参数，返回对应的 ASCII 数值。两个单字符之间的比较都会转化为对应的 ASCII 值之间的关系。例如：'a'<'b'，'a'>'A'，'0'<'1'。

2．多字符字符串的比较

（1）从两个字符串中索引为 0 的位置开始比较。

（2）比较位于当前位置的两个单字符。

如果两个字符相等，则两个字符串的当前索引加1，回到步骤（2）开始；如果两个字符不相等，返回这两个字符的比较结果，作为字符串比较的结果。

（3）如果两个字符串到一个字符串结束时都相等，那么较长的字符串更大。

【例 3.12】字符串的比较

```
#字符串的比较
a = 'abc'
b = 'abc'
print('a is b',a is b)
print('id(a) == id(b)',id(a) == id(b))
print('a==b',a==b)
c='ab'
d='abc'
print('c<d',c<d)
e='ac'
f='ab'
print('e<f',e<f)
g='hello word'
h='hello word'
print('g is not h',g is not h)
```

运行结果：

```
a is b True
id(a) == id(b) True
a==b True
c<d True
e<f False
g is not h False
```

3.5 字符串输入输出

计算机程序都是用来解决特定的计算问题的，每个程序都有统一的运算模式：输入数据、处理数据和输出数据。

3.5.1 字符串输入

python 使用 input 函数接收用户输入，其语法格式如下：

varname=input(“promptMessage”)

其中 varname 是 input()函数返回的字符串数据，promptMessage 是提示信息，其参数可以省略。当程序执行到 input()函数时，会暂停执行，等待用户输入，用户输入的全部数据均可以作为输入内容。如果要得到整数或小数，可以使用 exal()函数得到表达式的值，也可以使用 int()或 float()函数进行转换。

【例 3.13】字符串输入

```
#字符串输入
name=input('Please enter your name:')  #把接收到的值赋给 name 变量
print(name)    #输出接收到的输入
```

运行结果：

```
Please enter your name:Lucy
Lucy
```

3.5.2 字符串输出

Python3 使用 print()函数完成用户输出操作，print()函数的所有参数均可省略，如果没有参数，print()函数将输出一个空行。print()函数正常情况下均可以使用，可以使用一种包含一个字符串，字符串中可以包含另外一种（但是不可以包含同一种，否则需要转义。）

3.6 字符串运算


扫码看视频 3.5

字符串由若干个字符组成，为实现字符串的连接、子串的选择等，Python 提供了系列字符串的操作符，如表 3-4 所示，其中 a、b 是两个字符串，a=”hello”,b=” Python”。

操作符	描述	实例

+	字符串连接	a + b 输出结果： HelloPython
*****	重复输出字符串	a*2 输出结果：HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 **e**
[:]	截取字符串中的一部分，遵循左闭右开原则	a[1:4] 输出结果 **ell**
in	成员运算符，如果字符串中包含给定的字符返回 True	**'H' in a** 输出结果 True
not in	成员运算符，如果字符串中不包含给定的字符返回 True	**'M' not in a** 输出结果 True
r/R	原始字符串，它是用来替代转义符表示的特殊字符。原始字符串除在字符串的第一个引号前加上字母 **r**（R）以外，与普通字符串有着几乎完全相同的语法。	print(r'\n') print(R'\n') 输出：\n

表 3-4 python 字符串格式化控制符

表 3-4 中，+：连接符。+运算符需要两个字符串对象，连接起来得到一个新的字符串对象。

*：重复符。*运算符需要一个字符串对象和一个整数，新的字符串由原字符串复制而成，复制的次数为给出的整数值。

+运算符和*运算符都产生了新的字符串对象，但都不会影响表达式中的字符串。执行连接操作时，除非明确指出，否则在第一个字符串的末尾和第二个字符串开头位置之间没有空格。执行连接操作时，两个字符串对象的顺序是，第一个字符串显示在新的字符串对象的开始，第二个字符串在第一个字符串结束时开始。若更改顺序，则新的字符串对象中的顺序也发生改变。每个运算符所需要的操作对象的类型是特定的。对于连接操作，需要两个字符串对象。而复制操作，只需要一个字符串和一个整数，其他任何类型的组合都不能正常运行。

【例 3.14】字符串输出

```
#字符串输出
```

```
a = "Hello"
b = "Python"
print(a + b)
print(a * 2)
print(a[1])
print(a[1:4])
```

运行结果：

```
HelloPython
HelloHello
e
ell
```

3.7 字符串内建方法

字符串内建方法是从 python1.6 到 2.0 慢慢加进来的——它们也被加到了 Python 中。

这些方法实现了 string 模块的大部分方法，如下表 3-5 所示，列出了目前字符串内建支持的方法，所有的方法都包含了对 Unicode 的支持，有一些甚至是专门用于 Unicode 的。

操作符	描述
string.capitalize()	把字符串的第一个字符大写
string.center(width)	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
string.decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是 'ignore' 或者'replace'
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string))	检查字符串是否以 obj 结束，如果 beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8
string.find(str, beg=0, end=len	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查

操作符	描述
(string))	是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.format()	格式化字符串
string.index(str, beg=0, end=len(string))	跟 find()方法一样，只不过如果 str 不在 string 中会报一个异常.
string.isalnum()	如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False
string.isalpha()	如果 string 至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False
string.isdecimal()	如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit()	如果 string 只包含数字则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric()	如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle()	如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width)	返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string.lower()	转换 string 中所有大写字符为小写.
string.lstrip()	截掉 string 左边的空格
string.maketrans(intab, outtab])	maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str)	返回字符串 *str* 中最大的字母。
min(str)	返回字符串 *str* 中最小的字母。

操作符	描述
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把 字 符串 string 分 成 一 个 3 元 素 的 元 组 (string_pre_str,str,string _post_str),如果 string 中不包含 str 则 string_pre_str == string.
string.replace(str1, str2, num =string.count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len (string))	类似于 find()函数，不过是从右边开始查找.
string.rindex(str, beg=0,end=len(string))	类似于 index()，不过是从右边开始.
string.rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找
string.rstrip()	删除 string 字符串末尾的空格.
string.split(str="", num=string.count(str))	以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 num+ 个子字符串
string.splitlines([keepends])	按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果 beg 和 end 指定值，则在指定范围内检查.
string.strip([obj])	在 string 上执行 lstrip()和 rstrip()
string.swapcase()	翻转 string 中的大小写
string.title()	返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="")	根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 del 参数中
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符串 string 右对齐，前面填充 0

表 3-5 python 字符串内建方法

【例 3.15】字符串 Python count() 方法

```
#Python count() 方法用于统计字符串里某个字符出现的次数。
str = "this is string example....wow!!!"
sub = "i"
print("str.count(sub, 4, 40)",str.count(sub,4,40))
sub = "wow"
print("str.count(sub):",str.count(sub))
```

运行结果：

```
str.count(sub, 4, 40) 2
str.count(sub): 1
```

【例 3.16】字符串大小写转换的方法

```
#字符串大小写转换方法
str1="hello,Python"
print("str.lowes(str1):",str1.lower())
print("str.upper(str1):",str1.upper())
print("str.capitalize(str1):",str1.capitalize())
print("str.swapcase(str1):",str1.swapcase())
```

运行结果：

```
str.lowes(str1): hello,python
str.upper(str1): HELLO,PYTHON
str.capitalize(str1): Hello,python
str.capitalize(str1): HELLO,pYTHON
```

【例 3.17】查找和替换函数

```
#查找和替换函数
str1="hello,python"
print(str1.find("hello"))
print(str1.index("y"))
print(str1.replace("hello","hi"))
```

运行结果：

```
0
7
hi,python
```

3.8 实验

【例 3.18】python 中输入一个字符串，判断这个字符串中有多少个字符、数字、空格、特殊字符

```
# 判断这个字符串中有多少个字符、数字、空格、特殊字符
#首先定义一个字符串
str1 =input('请输入一个字符串:')
#初始化字符、数字、空格、特殊字符的计数
str_sum = 0
dig_sum = 0
spa_sum = 0
other_sum = 0
for strs in str1:
    #如果在字符串中有字符，那么字符的数量+1
    if strs.isalpha():
        str_sum += 1
    #如果在字符串中有数字，那么数字的数量+1
    elif strs.isdigit():
        dig_sum += 1
     #如果在字符串中有空格，那么空格的数量+1
    elif strs == ' ':
        spa_sum += 1
     #如果在字符串中有特殊字符那么特殊字符的数量+1
    else:
        other_sum += 1
print("该字符串中的字符有:",str_sum)
print("该字符串中的数字有:",dig_sum)
print("该字符串中的空格有:",spa_sum)
print("该字符串中的特殊字符有:",other_sum)
```

运行结果：

```
请输入一个字符串:sjfijlkmf345u843of9     @%%&o0mp983&
该字符串中的字符有: 15
该字符串中的数字有: 11
该字符串中的空格有: 3
该字符串中的特殊字符有: 5
```

【例 3.19】输入身份证号码输出对应的出生年月日

```
#输入身份证号码输出对应的出生年月日
ID = input('请输入十八位身份证号码: ')
if len(ID) == 18:
    print("你的身份证号码是 " + ID)
else:
    print("错误的身份证号码")
year = ID[6:10]
moon = ID[10:12]
day = ID[12:14]
print("出生年月: " + year + '年' + moon + '月' + day + '日')
```

运行结果：

```
请输入十八位身份证号码: 531624199810282860
你的身份证号码是 531624199810282860
出生年月: 1998 年 10 月 28 日
```

【例 3.20】字符串拆分与合并函数

```
#字符串拆分与合并函数
str1="hello,python,hello,c"
print(str1.split() )            #默认使用空格做分配符，str1 中无空格，列表中只有一个元素
print(str1.split(",") )          #使用逗号做分配符
print(str1.split(",",2) )       #使用逗号做分配符,限制分隔 2 次
lst=['hello','python!','hello','c!']
s=" "
print(s.join(lst))                    #将列表连接为字符串
```

运行结果：

```
['hello,python,hello,c']
['hello', 'python', 'hello', 'c']
['hello', 'python', 'hello,c']
hello python! hello c!
```

【例 3.21】字符串切片操作

```
#字符串切片操作
str1="hello,python!hello,c!"
print(str1[6:12])
print(str1[-8:-1])      #从后向前切片，最后一个字符索引是-1
print(str1[:-3])        #从索引为-3 的字符到字符串联
print('java' in str1)
print('python' in str1)
```

运行结果：

```
python
hello,c
hello,python!hello
False
True
```

第四单元：Python 的组合数据类型

Python 除整数类型、浮点数类型等基本的数据类型外，还提供了列表、元组、字典、集合等组合数据类型。组合数据类型能将不同类型的数据组织在一起，实现更复杂的数据表示或数据处理功能。

4.1 组合数据类型概述

根据数据之间的关系，Python 的组合数据类型可以分为三类：序列类型、映射类型和集合类型。

序列类型是 Python 中最基本的数据结构。序列中的每个元素都分配一个数字-它的位置，或索引，第一个索引是 0，第二个索引是 1，依此类推。序列都可以进行的操作包括索引、切片、加、乘、检查成员。此外，Python 已经内置确定序列的长度以及确定最大和最小的元素的方法。

序列包括字符串、列表和元组 3 种，字符串可以看作是单一字符的有序组合，属于序列类型。由于字符串类型十分常用且单一字符串只能表达一个含义，也被看作是基本的数据类型。列表和元组将在下面的章节进行介绍。

映射类型用键值对表示数据，典型的映射类型是字典；集合类型的数据中元素是无序的，集合中不允许有相同的元素存在。

4.2 列表


扫码看视频 4.1

列表是最常用的 Python 数据类型，它可以作为一个方括号内的逗号分隔值出现。列表的数据项不需要具有相同的类型。创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。

4.2.1 创建列表

列表是一种序列类型，标记“[]”可以创建列表，使用序列的常

用操作符可以完成列表的切片、检索、计数等基本操作。与字符串的索引一样，列表索引从 0 开始。列表可以进行截取、组合等。如下所示。

```
List1=[1,2,3,4,5]
```

其中列表 list1 中包含 5 个元素，分别是 1、2、3、4、5，list1 为列表名。这种创建列表的方式适用于对于列表中元素个数及其数值已知时。

【例 4.1】创建列表

```
#创建列表
list1 =[]
list2 = ['python', 'chemistry', 1997, 2000]
list3 = [1, 2, 3, 4, 5 ]
list4 = ["a", "b", "c", "d"]
print("list1=",list1)
print("list2[1]:",list2[1])
print("list3[1:4]:",list3[1:4])
print("list4[0]:",list4[0])
```

运行结果：

```
list1= []
list2[1]: chemistry
list3[1:4]: [2, 3, 4]
list4[0]: a
```

4.2.2 删除列表元素

可以使用 del 语句来删除列表的元素，如下实例：

【例 4.2】删除列表元素

```
#删除列表元素
list1= ['physics', 'chemistry', 1997, 2000]
print("list1:",list1)
del list1[2]
print("After deleting value at index 2 : ",list1)
```

运行结果：

```
list1: ['physics', 'chemistry', 1997, 2000]
```

```
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
```

4.2.3 修改列表元素

和字符串不同的是，列表是可变的，可以在列表中指定下标的值对元素进行修改。

【例 4.3】替换列表元素

```
#替换列表元素
lst=[12,13]
lst[1]=14
print('lst=',lst)
```

运行结果：

```
lst= [12, 14]
```

4.2.4 读取列表元素

元素下标表示该元素在 list 中的位置。注意 list 中元素下标是从 0 开始的，如第 n 个元素下标为 n-1。但当读取元素传入的元素下标超出 list 集合的大小时将会报“元素下标超出范围”的错误。

【例 4.4】读取列表元素

```
#读取列表元素
List1=["a","b","c","d"]
print('list1[0]=',List1[0])      #访问列表的第一个元素
print('list1[1]=',List1[1])      #访问列表的第二个元素
print('list1[5]=',List1[5])      #超出列表元素下标，报错
```

运行结果：

```
list1[0]= a
list1[1]= b
  File "D:/工作/python/案例源代码/第四单元/ex4.3.py", line 5, in <module>
    print('list1[5]=',List1[5])      #超出列表元素下标，报错
IndexError: list index out of range
```

由于 **list1** 的长度为 4，在取第 5 个元素时，**list1** 中元素的最大下标为 3<5，因此出现“list index out of range”的错误。

除了正向取 list 中的元素外，也可以逆向去取，用元素下标-1 表

示最后一个元素，-2 表示倒数第二个元素，同样注意不能超出元组个数的界限。

Python 的列表截取与字符串操作类型，代码如下所示：

L = ['spam', 'Spam', 'SPAM!']

表达式	描述	结果
L[2]	读取列表中第三个元素	'SPAM!'
L[-2]	读取列表中倒数第二个元素	'Spam'
L[1:]	从第二个元素开始截取列表	['Spam', 'SPAM!']

表 4-1 Python 的列表截取与字符串操作类型

4.2.5 遍历列表

遍历列表可以逐个处理列表中的元素，通常使用 for 循环和 while 循环来实现。第一种遍历方法隐藏了列表的长度，操作较为便利，第二种遍历方法则使用 len()函数计算出列表 numbers 的长度后进行遍历操作，其中 range()函数返回的是从 0 到 numbers 长度的数值序列。

【例 4.5】用 for 循环遍历列表

```
#遍历列表
lst=['primary school','secondary school','high school','college']
for item in lst:
    print(item,end=",")
```

运行结果：

```
primary school,secondary school,high school,college,
```

使用 while 循环遍历列表，需要先获取列表的长度，将获得的长度作为循环的条件。

【例 4.6】用 while 循环遍历列表

```
#用 while 循环遍历列表
#range()是创建一个初始值为 2，步长为 2，终值为 15 的整数列表
lst=list(range(2,15,2))
i=0
result=[]
while i<len(lst):
```

```
    result.append(lst[i]*lst[i])
    i+=1
    print(result)
```

运行结果：

```
[4]
[4, 16]
[4, 16, 36]
[4, 16, 36, 64]
[4, 16, 36, 64, 100]
[4, 16, 36, 64, 100, 144]
[4, 16, 36, 64, 100, 144, 196]
```

本实例首先构造一个初始值为 2，步长为 2，终值为 15 的列表，即[2,4,6,8,10,12,14]，然后在 while 循环中遍历，将计算得到的新值添加到空列表 result 中。

4.2.6 列表的方法

除了使用序号操作符操作列表，列表还有特有的方法，它们的主要功能完成列表元素的增删改查等。

Python 列表操作的函数和方法，列表操作包含以下函数:

操作符	描述
1cmp(list1, list2)	比较两个列表的元素
len(list)	列表元素个数
max(list)	返回列表元素最大值
min(list)	返回列表元素最小值
list(seq)	将元组转换为列表

表 4-2 Python 的列表操作的函数和方法

列表操作包含以下方法:

操作符	描述
list.append(obj)	在列表末尾添加新的对象
list.count(obj)	统计某个元素在列表中出现的次数
list.extend(seq)	在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
list.index(obj)	从列表中找出某个值第一个匹配项的索引位置
list.insert(index, obj)	将对象插入列表
list.pop(obj=list[-1])	移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
list.remove(obj)	移除列表中某个值的第一个匹配项

list.reverse()	反向列表中元素
list.sort([func])	对原列表进行排序

表 4-3 Python 的列表操作方法

增加列表元素有三种方法，具体如下：

方法一：使用“+”将一个新列表附加在原列表的尾部。列表是可变的，可以在列表中指定下标的值对元素进行修改。

方法二：使用 append()方法向列表的尾部添加一个新元素。

方法三：使用 extend()方法将一个列表添加在原列表的尾部。

【例 4.7】增加列表元素

```
#增加列表元素
list=[1]
list=list+['x', 'y']
print(list)
list.append(True)
print(list)
list.extend(['z',5])
print(list)
```

运行结果：

```
[1, 'x', 'y']
[1, 'x', 'y', True]
[1, 'x', 'y', True, 'z', 5]
```

【例 4.8】计算元素出现的次数

```
#计算元素出现的次数
li=[11,22,33,'aa','bb','cc','aa']
li_new=li.count("aa")
print(li_new)
```

运行结果：

```
2
```

4.3 元组

元组是包含 0 个多个元素的不可变序列

扫码看视频 4.2

类型，元组生成后是固定的，其中任意元素都不能被替换或删除。Python 的元组与列表类似，不同之处在于元组的元素不能修改。元组使用小括号，列表使用方括号。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

4.3.1 创建元组

元组通常使用标记”()”创建，创建一个空元组，代码如下：

```
tup = ()        #创建一个空元组
```

元组中只包含一个元素时，需要在元素后面添加逗号，否则括号会被当作运算符使用

【例 4.9】创建元组

```
#创建元组
student= (1, "tom", "2008-05-06", 10, 135.7)
num=(1, 2, 3, 4, 5, 6, 7 )
tup=((1,2,3),(4,5),(6,7),9)
print("student[0]: ", student[0])
print("num[1:5]: ", num[1:5])
print("tup:",tup[1])
```

运行结果：

```
student[0]:   1
num[1:5]:   (2, 3, 4, 5)
tup: (4, 5)
```

4.3.2 删除元组

上面说过，元组里的元素是不能修改的，所以也不能将元组中的某个元素删除，但是，可以将整个元组删除。

【例 4.10】删除元组

```
#删除元组
student= (1, "tom", "2008-05-06", 10, 135.7)
del student
print(student)
```

运行结果：

```
Traceback (most recent call last):
  File "D:/工作/python/案例源代码/第四单元/ex4.10.py", line 4, in <module>
    print(student)
NameError: name 'student' is not defined
```

此时，如果再调用 student，编译器将会抛出异常信息；提示该变量已经找不到了（未定义）。

4.3.3 修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合。但当元组中有列表时，可对元组中的列表进行修改。

【例 4.11】修改元组

```
#修改元组
tup1=(19,78.56)
tup2=('ab', 'xy')
tup3=(1,3,5,7,9,[1,3,5,7,9])
#  以下修改元组元素操作是非法的。
# tup1[0] = 100;
#  创建一个新的元组
tup4=tup1 + tup2
tup3[5][0]=-1
print(tup4)
print(tup3)
```

运行结果：

```
(19, 78.56, 'ab', 'xy')
(1, 3, 5, 7, 9, [-1, 3, 5, 7, 9])
```

4.3.4 元组运算符

与字符串、列表一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式	描述	结果
len((1, 2, 3))	计算元素个数	3
(1, 2, 3) + (4, 5, 6)	连接	(1, 2, 3, 4, 5, 6)
('Hi!',) * 4	复制	('Hi!', 'Hi!', 'Hi!', 'Hi!')
3 in (1, 2, 3)	元素是否存在	True
for x in (1, 2, 3): print x,	迭代	1 2 3

表 4-4 Python 的元组运算符

4.3.5 元组与列表的转换

元组与列表非常相似，只是元组中的元素值不能被修改。如果想要修改其元素值，可以将元组转换为列表，修改完后，再转换为元组。列表和元组相互转换的函数是 tuple(lst)和 list(tup),其中的参数分别为被转换对象。

【例 4.12】元组与列表的相互转换

```
#元组与列表的相互转换
tup1=(357,'abc','python','xyz')
lst1=list(tup1)
lst1.append(246)     #在列表末尾添加新的对象
tup1=tuple(lst1)
print(tup1)
```

运行结果：

```
(357, 'abc', 'python', 'xyz', 246)
```

4.4 字典

在程序设计中，存储成对的数据是十分常见的需求。例如，如果想要统计一篇英文文章时，各个单词的出现次数。在这种情况下，如果有一种数据结构，能够成对地存放单词和对应的次数，就会对完成单词次数统计的任务很有帮助。字典正是这样的数据类型，它是 Python 中内置的映射类型。映射是通过键值查找一组数据值信息的过程，由 key-value 的键值对组成，通过 key 可以找到其映射的值 value。

4.4.1 创建字典

字典是用于存储成对的元素，在一个字典对象中，键值不能重复，用于唯一标识一个键值对，

扫码看视频 4.3

而对于值的存储则没有任何限制。字典可以看作是由键值对构成的列表，在搜索字典时，首先查找键，当查找到键后就可以直接获取该键对应的值，这是一种高效实用的查找办法。

字典可以用标记“{}”创建，字典中每个元素都包含键和值两部分，键和值用冒号分开，元素之间用逗号分隔。dict()用于创建字典的函数，函数参数为“键=值”或“（键，值）”，参数中间用逗号隔开。

【例 4.13】创建字典

```
#创建字典
dict1={}
dict2={'yuwen':88,'shuxue':75,'yingyu':96}
dict3=dict(yuwen=88,shuxue=75,yingyu=96)
dict4=dict([('yuwen',88),('shuxue',75),('yingyu',96)])
print('dict1=',dict1)
print('dict2=',dict2)
print('dict3=',dict3)
print('dict4=',dict4)
```

运行结果：

```
dict1= {}
dict2= {'yuwen': 88, 'shuxue': 75, 'yingyu': 96}
dict3= {'yuwen': 88, 'shuxue': 75, 'yingyu': 96}
dict4= {'yuwen': 88, 'shuxue': 75, 'yingyu': 96}
```

第一行用于创建一个空的字典，该字典不包含任何元素，可以向字典中添加元素。

第二行是典型的创建字典的方法，是用“{}”括起来的键值对。

第三行使用 dict()函数，通过关键字参数创建字典。

第四行使用 dict()函数，通过键值对序列创建字典。

4.4.2 查找与反向查找字典元素

字典定义好后，可以通过键来查找值，这个操作称为“查找”。

【例 4.14】查找字典元素

```
#查找字典元素
d1=dict({"id":19,"name":"Marry","city":"chongqing"})
print(d1['id'])
print(d1["name"])
```

运行结果：

```
19
Marry
```

4.4.3 遍历字典

用循环遍历字典语句来遍历字典中的每个元素的键和值。

【例 4.15】查找字典元素

```
#遍历字典
d1=dict({"id":19,"name":"Lucy","city":"chongqing"})
for key in d1.keys():
        print(key,d1[key])
```

运行结果：

```
id 19
name Lucy
city chongqing
```

4.4.4 添加和修改字典元素

字典的大小和列表都是动态的，即不需要事先指定其容量大小，可以随时向字典中添加新的键—值对，或者修改现有键所关联的值。添加和修改的方法相同，都是使用“字典变量名[键名]=键值”的形式，主要区分在于字典中是否已存在该键—值对，若存在则为修改，否则为添加。

【例 4.16】添加和修改字典元素

```
dict1={'sex': 'female', 'name': 'Lucy', 'id': 19, 'city': 'kunming'}
#修改字典元素
dict1["id"]="18"
print('dict1=',dict1)
#添加字典元素
dict1["email"]="python@126.com"
print('dict1=',dict1)
```

运行结果：

```
dict1= {'sex': 'female', 'name': 'Lucy', 'id': '18', 'city': 'kunming'}
dict1= {'sex': 'female', 'name': 'Lucy', 'id': '18', 'city': 'kunming', 'email': 'python@126.com'}
```

4.4.5 检索字典元素

使用 in 运行符来测试某个特定的键是否在字典中。表达式为

```
key in dicts
```

其中，dicts 是字典名，key 是键名。

【例 4.17】检索字典元素

```
#使用 in 运算符检索
d1=dict({"id":19,"name":"Marry","city":"chongqing"})
if   "id" in d1:
     print("键 id 存在")
else :
     print("键 id 不存在")
```

运行结果：

```
键 id 存在
```

4.4.6 字典的常用函数

Python 字典的常用函数，如表 4-5 所示。

操作符	描述
dict.keys()	返回所有的键信息
dict.values()	返回所有的值信息
dict.items()	返回所有的键值对
dict.get(key,default)	键存在则返回相应值，否则返回默认值
dict.pop(key,default)	键存在则返回相应值，同时删除键值对，否则返回默认值
dict.popitem()	随机从字典中取出一个键值对，以元组（key,value）的形式返回
dict.clear()	删除所有的键值对
del dict[key]	删除字典中的某个键值对
dict.copy()	复制字典
dict.update(dict2)	将一个字典中的值更新到另一个字典中

表 4-5 Python 的常用函数

【例 4.18】get()函数

```
#get()函数
dict={'Name':'lucy','Age':19}
print("Name is:",dict.get('Name')+"\n"+ "Age is:",dict.get('Age'))
```

运行结果：

```
Name is: lucy
Age is: 19
```

4.5 集合


扫码看视频 4.4

集合是 0 个或多个元素的无序组合，集合与元组、列表类似，用于存储一系列的元素。可以很容易地向集合中添加元素或移除集合中的元素。集合中的元素只能是整数、浮点数、字符串等基本的数据类型。集合在的任何元素都没有重复的，这是集合的一个重要特点。集合与字典有一定的相似之处，但集合只是一组 key 的集合，这些 key 不可以重复，集合中没有 value。

4.5.1 创建集合

使用函数 set()可以创建一个集合。与列表、元组、元素等数据结构不同，创建集合没有快捷方式，必须使用 set()函数。set()函数最多有一个参数，如果没有参数，则会创建一个空集合。如果有一个参数，那么参数必须是可迭代的类型。例如字符串或列表，可迭代对象的元素将生成集合的成员。

【例 4.19】创建集合

```
#没有参数，set 会创建空集。
nullSet=set()
print('nullSet=',nullSet)
#提供一个 str 作为输入集合，创建一个 set。
a_set=set('abcd')
print('a_set=',a_set)
#提供一个 list 作为输入集合，创建一个 set。
s = set([1, 2, 3])
print('s=',s)
#重复元素在 set 中自动被过滤。
```

```
b = set([1, 1, 2, 2, 3, 3])
print('b=',b)
```

运行结果：

```
nullSet= set()
a_set= {'b', 'c', 'a', 'd'}
s= {1, 2, 3}
b= {1, 2, 3}
```

4.5.2 删除集合元素

删除集合元素有三种方法，具体如下：

一是通过 remove(key)方法删除元素。如果删除的元素不在集合中，remove 会报错。

二是 discard(key)删除元素，不同的是，如果删除的元素不在集合中，discard 不会报错。

三是 clear()删除集合的所有元素（使它成为空集）。

【例 4.20】删除集合元素

```
#删除集合元素
a=set(['y', 'python', 'b', 'o'])
a.remove('python')
print('a=',a)
b=set(['b', 'h', 'o', 'n', 'p', 't', 'y'])
b.discard('h')
print('b=',b)
c=set(['n', 'p', 't', 'y'])
c.clear()
print('c=',c)
```

运行结果：

```
a= {'y', 'o', 'b'}
b= {'o', 'b', 'p', 't', 'y', 'n'}
c= set()
```

4.5.3 添加集合元素

集合的添加有两种常用方法，分别是 add 和 update。可以添加元素到 set 中，可以重复添加，但不会有效果。

一是集合 add 方法：是把要传入的元素做为一个整个添加到集合中。

二是集合 update 方法：是把要传入的元素拆分，做为个体传入到集合中。

【例 4.21】添加集合元素

```
#添加集合元素
a = set('boy')
a.add('python')
print('a=',a)
b    = set('boy')
b.update('python')
print('b=', b)
```

运行结果：

```
a= {'b', 'o', 'y', 'python'}
b= {'b', 't', 'y', 'o', 'p', 'n', 'h'}
```

4.5.4 集合的遍历

用循环遍历集合语句来遍历集合中的每个元素。

【例 4.22】集合的遍历

```
#集合的遍历
a=set("python")
for x in a:
      print(x,end=" ")
```

运行结果：

```
h t n y o p
```

4.5.5 集合运算

Python 提供了众多内置操作集合的方法，用于添加集合元素、复制集合、统计元素个数等等。Python 中的集合与数学中集合的概念是一致的，因此，两个集合可以做数学意义上的交集、并集、差集计算等。

方法	描述
S.copy()	复制集合。
S.pop()	随机集合 S 中的一个元素，并在集合中删除该元素。S 为空时产生 KeyError 异常
S.isdisjoint(T)	判断集合中是否存在相同元素。如果集合 S 和 T 没有相同元素，则返回 Ture
len(S)	返回集合 S 的元素个数
S&T 或 S.intersaction(T)	交集。返回一个新集合，包括同时在集合 S 和 T 中的元素
S|T 或 S.union(T)	并集。返回一个新集合，包括集合 S 和 T 中的所有元素
S-T 或 S.difference(T)	差集。返回一个新集合，包括集合 S 中但不在集合 T 中的所有元素
S^T 或 S.symmetric_difference_update(T)	补集。返回一个新集合，包括集合 S 和 T 中的所有元素，但不包括同时在其中的元素
S<=T 或 S.issubset(T)	子集测试。如果 S 和 T 相同或 S 是 T 的子集，返回 Ture，否则返回 False
S>=T 或 S.issuperset(T)	超集测试。如果 S 和 T 相同或 S 是 T 的超集，返回 Ture，否则返回 False

表 4-6 Python 集合运算符

【例 4.23】集合运算

```
#集合的运算
a=set([10,20,30])
b=set([20,30,40])
set1=a&b
set2=a|b
set3=a-b
set4=a^b
set5=set1<a
set6=a<set2
print('set1=',set1)
print('set2=',set2)
print('set3=',set3)
print('set4=',set4)
print('set5=',set5)
print('set6=',set6)
```

运行结果：

```
set1= {20, 30}
set2= {20, 40, 10, 30}
set3= {10}
set4= {40, 10}
set5= True
set6= True
```

4.6 实验

【例 4.24】英文歌曲中的词频统计

词频统计需要考虑以下几个方面：

1.英文单词的分隔符可以是空格、标点符号或特殊符号，使用字符串的 replace()方法可以将以标点符号替换成空格，以提高获取单词的准确性。

2.用 split()函数可以拆分字符串，生成单词的列表。

3.逐个读取列表中的单词，并重复以下的操作。

如果字典的 dict1 值中没有这个单词，向字典中添加元素，关键字是这个单词，值是 1，即 dict1[word]=1;如果字典的 key 值中有这个单词，则该单词计数加 1，即 dict1[word]+=1；当列表中的单词全部读取完后，每个单词出现的次数会被放在字典 dict1 中，dict1 的 key 是单词，value 是单词出现的次数。

```
#英文歌曲中的词频统计
lyrics ='Dashing thro the snow, in a one-horse open sleigh.Over the fields we go, laughing all the way.\
Bells on bob-tails ring, making spirits bright,What fun it is to ride and sing a sleighing song tonight.\
Jingle bells, jingle bells, jingle all the way. Oh what fun it is to ride in a one horse open sleigh.\
Jingle bells, jingle bells, jingle all the way. Oh what fun it is to ride in a one horse open sleigh. '
#将歌词中涉及的标点用空格替换
for ch in ",.?!":
    lyrics=lyrics.replace(ch," ")
#利用字典统计词频
words=lyrics.split()
dict1={}
for word in words:
    if word in dict1:
        dict1[word]+=1
```

```
        else:
            dict1[word]=1
#对统计结果排序
items=list(dict1.items())
items.sort(key=lambda x:x[1],reverse=True)
#打印控制
for item in items:
    word,count=item
    print("{:<12}{:>5}".format(word,count))
```

运行结果：

```
the          5
a            4
bells        4
jingle       4
in           3
open         3
sleigh       3
all          3
way          3
fun          3
it           3
is           3
to           3
ride         3
Jingle       2
Oh           2
what         2
one          2
horse        2
Dashing      1
thro         1
snow         1
one-horse    1
Over         1
fields       1
we           1
go           1
laughing     1
Bells        1
on           1
bob-tails    1
ring         1
making       1
```

```
spirits          1
bright           1
What             1
and              1
sing             1
sleighing        1
song             1
tonight          1
```

第五单元：Python 程序的流程控制

在解决实际问题时，我们会遇到根据不同的条件来选择不同的操作，或者经常会遇到需要重复处理相同或相似的操作。在 python 中，可以使用一些判断和循环语句去解决这些问题。本章首先介绍一些判断语句，包括简单的 if 语句、if-else 语句、if-elif-else 语句，然后介绍 while 循环和 for 循环语句。

5.1 程序设计流程

python 设计程序一般分为如下步骤。步骤 1：分析找出解决问题的关键之处，即找出解决问题的算法，确定算法的步骤。步骤 2：将算法转换为程序流程图。步骤 3：根据程序流程图编写符合 python 语法的代码。步骤 4：调试程序，纠正错误。

在 python 中，提供了专门的控制语句来控制程序的执行，这些语句称为流程控制语句。常见的流程为顺序结构，选择结构，循环结构。

5.2 顺序结构

顺序结构是最简单的控制结构，按照语句的书写顺序依次从上到下执行，如图 5-1 是一个顺序结构的流程图，它有一个入口、一个出口，依次执行语句 1 和语句 2。

一般情况下，实现程序顺序结构的语句主要是赋值语句和内置的input()输入函数和 print()输出函数。这些语句可以完成输入、计算、输出的基本功能。

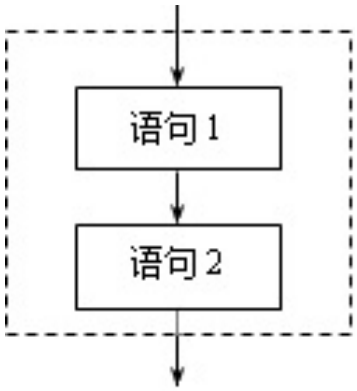


图5-1 顺序结构的流程图

【例 5.1】顺序结构案例。编写程序，要求输入三角形的三条边（假设给定的三条边符合构成三角形的条件：任意两边之和大于第三边），计算三角形的面积并输出。

提示：三角形面积公式 area = (s * (s-a) * (s-b) * (s-c)) ** 0.5，其中 s = (a + b + c)/2。

程序代码：

```
a = input("请输入三角形的第一条边长：")    #输入第一条边长
b= input("请输入三角形的第二条边长：")    #输入第二条边长
c= input("请输入三角形的第三条边长：")    #输入第三条边长
a,b,c = int(a),int(b),int(c)    #将输入的三条边长分别转换为整型
s = (a + b + c)/2      #计算 s
area = (s * (s-a) * (s-b) * (s-c)) ** 0.5      #计算面积
print("此三角形面积为：",area)   #计算三角形面积
```

运行结果：

```
请输入三角形的第一条边长：3
请输入三角形的第二条边长：4
请输入三角形的第三条边长：5
此三角形面积为：6.0
```

5.3 选择结构

选择结构又称为分支结构，即按照给定条件来选择其中一个分支执行程序中特定的语句。在 Python 语言中，选择结构分为：单选择结构（if 语句）、双选择结构（if...else 语句）和多选择结构（if...elif 语句）。

5.3.1 if 语句


扫码看视频 5.1

if 语句通过条件表达式来判断真假，当且仅当该表达式为真时，则执行语句序列，否则直接执行 if 语句下面的语句。if 语句的语法格式如下：

```
if <表达式>:
    <语句序列>
```

其中：if 为 Python 的关键字，<表达式>是任意的数值、字符、关系或逻辑表达式，或用其它数据类型表示的表达式。它表示条件，以 True 表示真，False 表示假。

紧跟在“”表达式：”后面的<语句序列>称为 if 语句的内嵌语句以缩进方式表达，缩进的<语句序列>可以是一条语句，也可以是多条语句，当有多条语句时，保持每条语句的缩进相同。编辑器也会提示程序员开始书写内嵌语句的位置，如果不再缩进，表示内嵌语句在上一行就写完了。执行顺序如图 5-2if 语句流程图。

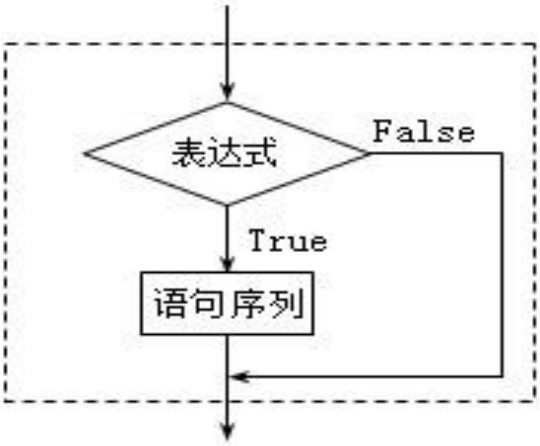


图5-2 if语句的流程图

【例 5.2】分支结构举例。输入两个整数 a 和 b，按从小到大的顺序输出这两个数。

分析：若 a>b，则将 a、b 交换，否则不交换。

程序代码：

```
a = int(input("a="))     #输入变量 a 的值并转换为整型
b =int(input("b="))     #输入变量 b 的值并转换为整型
print("before exchange:",a,b)     #输出交换前两个变量的值
if a>b:          #if 语句条件
      a,b=b,a     #if 语句块
print("after exchange:",a,b)    #if 结构外语句，该语句一定会执行
```

运行结果：

```
a=3
b=2
before exchange:3 2
after exchange:2 3
```

5.3.2 if...else 语句

扫码看视频 5.2

if...else 语句为双选择结构，当某个条件为真时，使用一个 if 语句会完成一个动作。而如果条件为 false 是时，程序将不执行任何动作而继续向后执行。if...else 语句的语法格式如下：

```
if <表达式>:
      <语句序列 1>
```

```
else:
    <语句序列 2>
```

执行顺序是：首先计算表达式的值，若<表达式>的值为 True，则执行<语句序列 1>，否则执行<语句序列 2>。执行顺序如图 5-3 if...else 语句流程图：

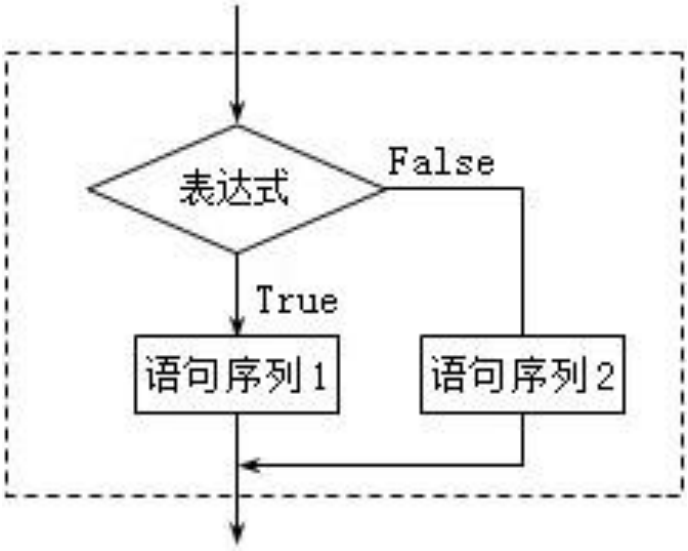


图5-3　if...else语句的流程图

【例 5.3】编写程序，输入一个学生成绩 grade 判断是否及格。

分析：本实例可用 if...else 语句进行判断，如果输入的学生成绩 grade>=60，输出“及格”，否则输出“不及格”。

```
grade=int(input("请输入学生成绩:"))  #输入变量 grade 的值并转换为整型
if grade>=60:    #判断 grade 是否大于或等于 60
    print("及格")  #如果是，输出“及格”
else:       #如果不是
    print("不及格")    #输出“不及格”
```

运行结果：

```
请输入学生成绩:70
及格
```

【例 5.4】编写程序，输入学生年龄，判断该学生是否成年。

分析：用 if-else 语句判断，如果输入的年龄小于 18 岁，输出“未成年”，否则输出“已成年”。

程序代码：

```
age = int(input("请输入学生的年龄："))  #输入变量 age 的值并转换为整型
if age<18:                          #判断 age 是否小于 18
    print("未成年")                      #如果是，输出“未成年”
else:                           #如果不是
    print("已成年")                      #输出“已成年”
```

运行结果：

```
请输入学生的年龄：18
已成年
```

【例 5.5】输入一个年份 year，判断是否为闰年。

分析：闰年的条件为：（1） 能被 4 整除但不能被 100 整除；（2）能被 400 整除。

程序代码：

```
year = int(input("输入年份："))
if (year%4==0 and year%100 !=0) or (year%400==0):
    print(year,"是闰年")
else:
    print(year,"不是闰年")
```

运行结果：

```
输入年份：2019
2019 不是闰年
```

5.3.3 if...elif...else 语句

扫码看视频 5.3

if...elif...else 语句为 pythton 中的多选择结构，当选择结构需要的分支多于两个时，就需要用到多分支结构。多分支结构只能根据条件的 True 和 False 决定处理哪个语句序列。if...elif...else 语句的语法格式：

```
if <表达式 1>:
    <语句序列 1>
elif <表达式 2>:
    <语句序列 2>
    …
elif <表达式 n>:
```

```
        <语句序列 n>
else:
        <语句序列 n+1>
```

执行顺序是：首先计算表达式 1 的值，若<表达式 1>的值为 True，则执行<语句序列 1>；否则将计算表达式 2 的值，若<表达式 2>的值为 True，则执行<语句序列 2>；否则计算<表达式 n>的值，若<表达式 n>的值为 True，则执行<语句序列 n>；若所有表达式的条件都不满足，则执行最后一个 else 后面的<语句序列 n+1>。如图 5-4if...elif...else 语句流程图：

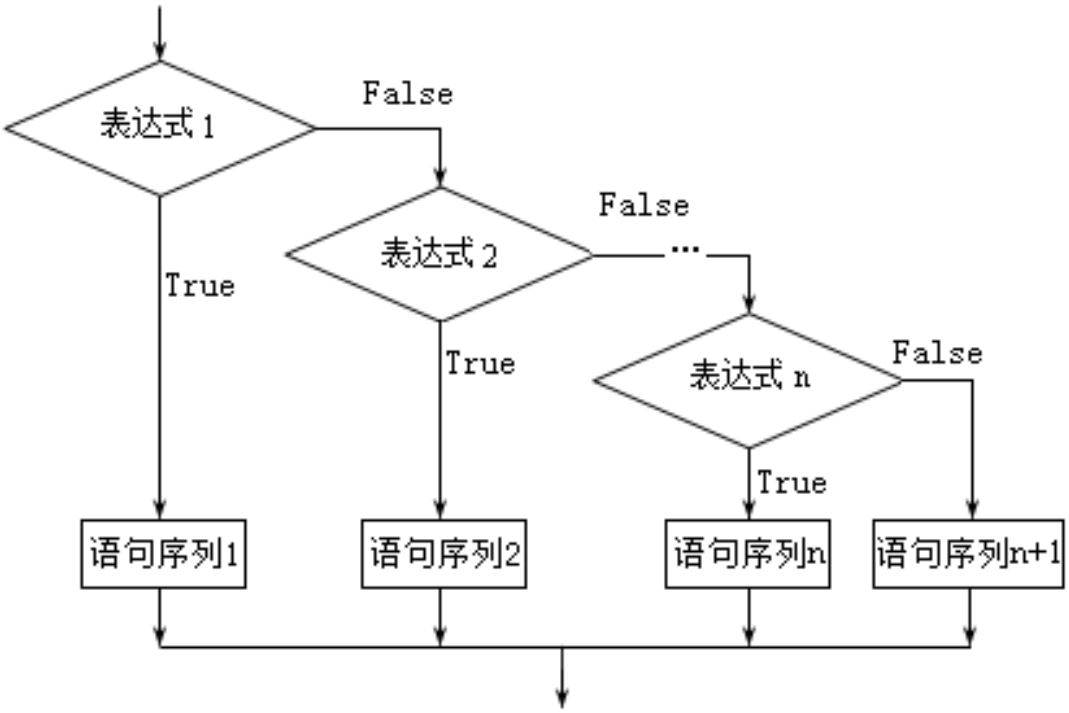


图5-4 if...elif...else语句流程图

值得注意的是：（1）不管有几个分支，当某个表达式满足条件时，将执行该表达式对应的分支，其余分支语句序列将不再执行。（2）当多分支中有多个表达式同时满足条件，则只执行第一条与之匹配的语句。

【例 5.6】学生成绩可分为百分制和五级制，将输入的百分制成

绩 score，转换成相应的五级制成绩后输出，百分制和五级制成绩的转换关系如表 5-1。

表5-1 百分制和五级制成绩的转换关系

百 分 制	五 级 制	百 分 制	五 级 制
90≤score≤100	A	60≤score < 70	D
80≤score < 90	B	0≤score < 60	E
70≤score < 80	C	score>100 或 score<0	无效

程序代码：

```
score=float(input("请输入百分制成绩："))#输入分数 score 的值并将其转化为浮点数
if score>100 or score<0:          #当分值不合理时显示出错信息
    print("输入数据无效")
elif score>=90:                   #当成绩大于等于 90 小于等于 100 时，输出“A”
    print("A")
elif score>=80:                   #当成绩大于等于 80 小于 90 时，输出“B”
    print("B")
elif score>=70:                   #当成绩大于等于 70 小于 80 时，输出“C”
    print("C")
elif score>=60:                   #当成绩大于等于 60 小于 70 时，输出“D”
    print("D")
else:                   #以上条件都不满足
    print("E")      #输出“E”
```

运行结果：

```
请输入百分制成绩：75
C
```

【例 5.7】计算某住户一年应交的天然气费。

表5-2 中石油昆仑燃气公司公布的天然气收费标准表

档次	年用气量	到户价格
第一档	年用气量<=360立方米	2.95
第二档	360立方米<年用气量<=540立方米	3.54
第三档	年用气量>540立方米	4.43

程序代码：

```
total=int(input("请输入年用水量："))
```

```
if total <=360:
        price=2.95*total
elif total<=540:
        price=2.95*360+3.54*(total-360)
else:
         price=2.95*360+3.54*(540-360)+4.43*(total-360)
print("年用气量为{0}立方米的用户需缴纳气费为{1}元".format(total,price))
```

运行结果：

```
请输入年用水量：540
年用气量为 540 立方米的用户需缴纳气费为 1699.2 元
```

5.4 循环结构

循环结构是一种让指定的代码块重复执行的有效机制，Python 可以使用循环使得在满足“预设条件”下，可以重复执行一段语句块。构造循环结构有两个要素，一是循环体，即重复执行的语句和代码，另一个是循环条件，即重复执行代码所要满足的条件。为了能够适应不同场合的需求，Python 用 while 和 for 关键字来构造两种不同的循环结构，即表达两种不同形式的循环条件。

5.4.1 while 语句

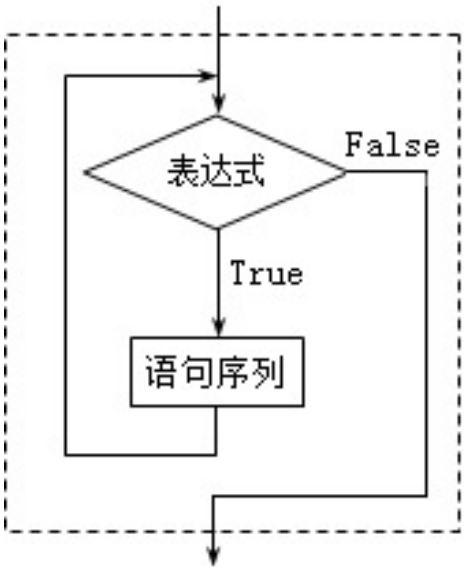
扫码看视频 5.4

while 语句用于实现循环结构，其特点是先判断，后执行。如果刚进入循环时条件就不满足，则循环体一次也不执行。还需要注意的是，一定要有语句修改判断条件，使其有为假的时候，否则将出现“死循环”。while 语法格式如下：

```
While <表达式> :
    <语句序列>
```

其中：<表达式>称为循环条件，可以是任何合法的表达式，其值为 True、False，它用于控制循环是否继续进行。<语句序列>称为循环体，它是要被重复执行的代码行。

执行顺序是：首先判断<表达式>的值，若值为真，则执行循环体<语句序列>，接着再判断<表达式>，直至<表达式>的值为 False 时退出循环 。如图 5-5while 语句流程图：



如图5-5 while语句流程图

【例 5.8】在屏幕上打印输出重复字符串。

程序代码：

```
i=0
while(i<3):
#循环开始
    i=i+1;
    print("python",i)
#循环结束
```

运行结果：

```
python 1
python 2
python 3
```

【例 5.9】数字累加求和。

编写程序，计算 1+2+3+...+50 的值。

分析：这是一个累加问题，需要将 50 个数相加。可以用 while 循环来实现，重复执行循环体 50 次，每次累加一个数。

程序代码：

```
i=1                    #创建变量 i，赋值为 1
Sum=0                  #创建变量 Sum，赋值为 0
while i<=50:           #循环，当 i>50 时结束
    Sum+=i             #求和，将结果放入 Sum 中
    i+=1               #变量 i 加 1
print("Sum=1+2+3+…+50= ",Sum)       #输出 Sum 的值
```

运行结果：

```
Sum=1+2+3+…+50=1275
```

5.4.2 for 语句

扫码看视频 5.5

在 Python 中，for 循环可以遍历任何序列的项目，如一个列表或者一个字符串。for 循环语句的语法结构如下：

```
for <变量> in <序列> :
        <语句序列>
```

其中，<变量>可以扩展为变量表，变量与变量之间用“,”分开。<序列>可以是序列、迭代器或其它支持迭代的对象。列表、元组、字符串以及字典等都属于序列型的对象。

for 语句的执行顺序：<变量>取遍<序列>中的每一个值。每取一个值，如果这个值在<序列>中，执行<语句序列>，返回，再取下一个值，再判断，再执行，依次类推，直到遍历完成或发生异常退出循环。

经常使用的遍历方式有：

- 有限次遍历：for i in range(n):　　#n 为遍历次数
- 遍历文件：for line in myfile:　# myfile 为文件的引用
- 遍历字符串：for ch in mystring:　#mystring 为字符串的引用
- 遍历列表：　for item in mylist:　#mylist 为列表的引用

【例 5.10】使用 for 循环输出字符串"ABC"中的每一个字符。

```
for i in "ABC":
  print(i)
```

运行结果：

```
A
B
C
```

字符串也是一种序列型对象，假设要统计用户输入的一段字符串中某个字符的数量，也可以通过 for 循环实现。

【例 5.11】统计用户输入的一段字符串中某个字符的数量

```
str=input("请输入一个语句：")
i=0
for ch in str:
      if ch==' ':
         i= i+1
print("这个语句有",i,"空格")
```

运行结果：

```
请输入一个语句：I love python
这个语句有 2 空格
```

【例 5.12】我们可以创建一个 ex.txt 文件存放在本地 D 盘,在 txt 中输入 I Love python!我们可以获取到 txt 文件里的内容。

程序代码：

```
fd=open("D:/ex.txt")
for i in fd.readlines():
      print(i)
```

运行结果：

```
I Love python!
```

【例 5.13】遍历文件

```
myfile = open("D:/ex.txt")
for line in myfile:
  print(line,end="")
```

运行结果：

```
I Love python!
```

【例 5.14】使用 for 循环求 1~10 的整数累加。

程序代码：

```
sum=0    #创建变量 sum,赋值为 0
for i in range(1,11):    #循环变量 i 从 1 取到 10
    sum+=i   #求和，将结果放入 sum 中
print("sum=1+2+3+....+10=",sum)    #输出 sum 的值
```

运行结果：

```
sum=1+2+3+....+10= 55
```

5.5 流程控制的其他语句

扫码看视频 5.6

pass 语句可以算作顺序语句，可用在任何地方，是为了保持程序结构的完整性。break、continue可以算作特别的顺序语句，而且它们用的地方特别。它们要与分支、循环语句合作使用。

1、pass 语句

pass 语句表示不做任何事情，一般用做占位语句，可以用在任何地方。 pass 语句语法格式如下：

```
pass      # 什么事也不做，一个空语句
```

【5.15】输出字符串"ABC"中的每一个字符

```
for i in "ABC":
 if i == 'B':
    pass
    print("这是 pass 块")
 print(i)
```

运行结果：

```
A
这是 pass 块
B
C
```

2、break 语句

break 语句用在循环语句（迭代）中，结束当前的循环（迭代）跳转到循环语句的下一条。break 语句常常与 if 语句联合，满足某条件时退出循环（迭代）。break 语句语法：

```
break
```

【例 5.16】break 语句实例

```
i = 10    #创建变量 i，赋值为 10
while i>0:   #创建 while 循环条件
  print("当前变量值:",i)
  i = i -1
  if i==5:   #  当变量 i 等于 5 时退出循环
        break     #退出循环
print("over!")   #输出 over
```

运行结果：

```
当前变量值: 10
当前变量值: 9
当前变量值: 8
当前变量值: 7
当前变量值: 6
over!
```

【例 5.17】break 语句对循环的作用

```
for i in range(1,5):
  print(i)
  if(i%3):
       print("*")
  else:
     break
  print("%")
print("over")
```

程序说明：循环在 i 为 1,2 时，执行 if 语句及其后的一条输出语句，当 i=3 时，执行 break 语句，循环结束，循环执行了 3 次，然后执行循环后语句，输出“over”

运行结果：

```
1
*
```

```
%
2
*
%
3
over
```

3、continue 语句

continue 语句用在循环语句（迭代）中，忽略循环体内 continue 语句后面的语句，回到下一次循环（迭代）。

在例 5.17 中，将 break 语句替换成 continue 语句。则会输出不同的结果：

```
for i in range(1,5):
  print(i)
  if(i%3):
      print("*")
  else:
      continue
  print("%")
print("over")
```

运行结果：

【例 5.18】 continue 语句输出 1~10 之间的所有奇数。

```
for i in range(1,11): #循环，i 的取值为 1 到 10
    if i%2==0:      #判断 i 是否为偶数
        continue     #当 i 为偶数时跳出本次循环
    else:            #当 i 为奇数时输出 i 的值
        print(i)
```

运行结果：

```
1
3
5
7
9
```

5.6 实验：判断奇偶数、数字累加求和、输出九九乘法表

本节通过一些实验，以加深对 Python 的流程控制的认识和使用。

5.6.1 判断奇偶数

分析：在 Python 中判断奇数偶数的条件是： 如果输入的数能被 2 整除的为偶数，不能被 2 整除的为奇数。

程序代码：

```
i=int(input("输入一个数字: "))     #输入一个数字并转化为整数
if (i%2) == 0:           #判断 i 是否为偶数
  print(i,"是一个偶数  ")     #如果 if 条件满足则执行该语句
else:
  print(i,"是一个奇数")       #如果 if 条件满足则执行该语句
```

运行结果：

```
输入一个数字:10
10 是一个偶数
```

5.6.2 求两个整数 m 和 n 的最大公约数

该算法的思想是：

① 对于已知两数 m,n，使得 m>n；

② m 除以 n 得余数 r；

③ 若 r≠0，则令 m←n，n←r，继续相除得新的 r；直到 r=0 求得最大公约数，结束。

程序代码：

```
a=int(input("请输入一个数:  "))
b=int(input("在输入一个数:  "))
x=a
y=b
if x<y:
      x,y=y,x
r=x%y
while r!=0:
            x=y
            y=r
            r=x%y
```

```
print(a,b,"的最大公约数是",y)
```

运行结果：

```
请输入一个数：6
在输入一个数：8
6 8 的最大公约数是 2
```

5.6.3 输出九九乘法表

编写程序，输出九九乘法表。

分析：可以使用 for 语句循环嵌套，外循环控制行，内循环控制列。

程序代码：

```
for j in range(1,10):              #循环变量 j 从 1 循环到 9
    for k in range(1,j+1):      #循环变量 k 从 1 循环到 j+1
        print(k,"*",j,"=",j*k,"",end="")#输出乘法表达式
    print("")                  #输出空字符串，作用是为了换行
```

运行结果：

```
1*1=1
1*2=2  2*2=4
1*3= 3  2*3=6   3*3=9
1*4=4   2*4=8   3*4=9  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72   9*9=81
```

第六单元：用函数实现代码复用

在实际开发过程中，经常会遇到一些重复或相似的操作，使用函数可以实现代码的复用，从而增加代码的可用性和可靠性。本章主要介绍函数的定义，函数的调用，参数的类型，返回值，通过案例，使

读者进一步了解函数在程序设计中的运用。

6.1 认识 Python 的函数

函数是一段具有特定功能的、可重复使用的代码段，它能够提高程序的模块化和代码的复用率。一个较大的程序，通常需要合理划分程序中的功能模块，功能模块在程序设计语言中被称为函数。要写好函数，必须清楚函数的组织格式（即函数如何定义）；要用好函数，则必须把握函数的调用机制。

使用函数有两个目的：（1）分解问题，将一个大问题或者大功能分解为多个小问题，从而降低编程难度，使问题更容易解决。（2）避免编写重复的代码。

Python 提供了很多系统内置函数（如 print()、input()、int()、ord()、len()函数等），系统内置函数是用户可直接使用的函数。标准库函数（如 math 库中的 sqrt()函数），要导入相应的标准库，才能使用其中的函数。用户自定义函数，用户还可以自己编写函数，只有定义了这个函数，用户才能调用。这是本章要讨论的问题。


扫码看视频 6.1

6.2 函数的定义和调用

在 python 中，使用 def 关键字来定义函数，而函数调用通过调用语句实现，调用语句所在的程序或函数称为调用程序或调用函数。

6.2.1 函数的定义

函数定义的一般格式为：

```
def 函数名(参数 1,参数 2, …,参数 n):
''文档字符串'''
```

```
    函数体
    return [返回值列表]
```

其中，函数名是任何有效的 Python 标识符，参数表是用“,”分隔的参数，参数个数可以是 0 个、1 个或多个，参数用于调用程序在调用函数时向函数传递值。函数体是函数被调用时执行的代码段，至少要有一条语句。

在定义函数时，需要注意的是：

（1）即使函数名后面的参数个数是 0 个，也必须保留一对空括号，且括号后面的冒号不能省略。

（2）文档字符串是用来告诉调用者这个函数的功能，可有可无。

（3）编写函数体时，函数体与 def 关键字之间需进行缩进。

（4）return 语句可以在函数体的任何位置出现，return 语句是可选的，如果有 return 语句，但是 return 后面没有接表达式或者值的，则返回 None；如果没有 return 语句，则会自动返回 None。

下面自定义一个名为“say_abc()”的函数，该函数的作用是输出“abc”字符串。

```
def say_abc():   #自定义一个无参数的函数
    print('abc')
```

6.2.2 函数调用

定义了函数后，就相当于有了一段具有特定功能的代码，在执行 def 时并不会执行这些代码（即函数体），想要执行这些代码，需要调用函数。调用函数需要指定被调用函数的名字和调用该函数参数。函数调用的格式为：

```
函数名（实参 1，实参 2，……）
```

其中，函数名——是事先定义函数时定义的函数名。参数表——

此时应是实际参数表，即实参表，实参之间用“,”分隔，实参要有确定的值。实参的个数可以少于形参的个数，这是由于形参有默认值。

【例 6.1】使用自定义的 say_abc()函数，输出“abc”字符串。

程序代码：

```
def say_abc():   #自定义一个无参数的函数
    print('abc')
say_abc()   #  调用函数
```

运行结果：

```
abc
```

【例 6.2】使用函数，实现下列内容输出。

****

这是一个函数调用实例

****

分析：输出的第 1,3 行相同，可以使用同一函数在合适的地方调用。在此例中，需定义两个函数，一个用于输出*号，一个用于输出“这是一个函数调用实例”。

程序代码：

```
def print_xing():   #定义名为 print_xing 的函数
    print("**********") #输出****
def print_text():   #定义 print_text 函数
    print("这是一个函数调用实例")
print_xing()          #调用函数
print_text()          #调用函数
print_xing()          #调用函数
```

运行结果：

```
**********
这是一个函数调用实例
**********
```

6.3 函数的参数和返回值

扫码看视频 6.2

6.3.1 函数的参数

在 python 中，参数的类型很多，有形式参数（形参），实际参数（实参），必备参数、关键字参数，默认参数，不定长参数。

1、形参与实参

形参：使用 def 语句来定义函数时，函数名后面的圆括号中的参数就是形式参数（简称形参）。形参只能是变量，只有函数被调用时才分配内存单元，调用结束时释放所分配的内存单元。

实参：调用函数时，函数名后面的圆括号中的参数。实参可以是常量、变量、表达式，在实施函数调用时，实参必须有确定的值。

【例 6.3】使用 max()函数，求两个数中值较大的数。

```
def max(a,b):       #定义 max 函数
   if a>b:     #如果条件成立，返回 a 的值
        return a
   else:       #否则返回 b 的值
      return b
x=int(input("输入一个数："))   #显示提示语并接收 x 的值
y=int(input("再输入一个数："))   #显示提示语并接收 y 的值
z=max(x,y)     #调用函数，将较大值赋给 z
print("较大的数为：",z)     #输出较大值
```

运行结果：

```
输入一个数：1
再输入一个数：3
较大的数为：  3
```

在此列中，max()函数括号内的 a 和 b 就是该函数的形参，而调用该函数时，括号内的 x 和 y 则是传递给该函数的实参。

2、必备参数

必备参数须以正确的顺序传入函数，调用函数时，输入的实参和形参的顺序要一致，实参数量必须和声明函数时形参的数量一样，不

然会出现语法错误：

【例 6.4】运行下列程序，分析结果。

```
def print_text(x, y):
    print(x,y)
print_text("hello","word")
print_text("hello")
```

运行结果：

```
hello word
Traceback (most recent call last):
  File "C:/Users/Administrator/Desktop/python 程序设计资料/第 6 章源代码/6.4.py", line 4, in <module>
    print_text("hello")
TypeError: print_text() missing 1 required positional argument: 'y'
```

结果分析：当执行语句 print_text("hello","word")时，实参和形参的数量相等，将实参”hello”,”word”分别传递给形参 x,y,实现输出。当执行语句 print_text("hello")时，实参和形参的数量不等，只传入了一个参数，程序运行到这条语句时就提示缺少要求的位置参数 y 的错误信息。

3、关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

【例 6.5】关键字参数实例。输出某商品的价格和折扣。

程序代码：

```
def fun(discount,price):              #定义函数
    print ("商品折扣: ",discount)
    print ("商品价格: ", price)
    return
fun(price=100,discount=0.5)          #关键字参数传递
```

运行结果：

```
商品折扣: 0.5
商品价格: 100
```

4、默认参数

在定义函数时，可以为函数的某个形参赋予默认值，这个参数被称为默认值参数。带有默认值参数的函数定义语法如下：

```
def  函数名(…,形参名=默认值):
        函数体
```

在调用带有默认值参数的函数时，可以不用为设置了默认值的形参进行传值，此时函数将会直接使用函数定义时设置的默认值，也可以通过显式赋值来替换其默认值。

【例 6.6】默认值参数。

```
def fun(discount,price=100):                    #定义函数
        print ("商品折扣: ",discount)
        print ("商品价格: ", price)
        return
fun(0.5)
```

程序运行结果：

```
商品折扣:   0.5
商品价格:   100
```

在调用 fun()函数时，只输入了一个实参值。从结果知道，尽管实参没有传入具体值，price 仍然会按定义时的默认值输出。

值得注意的是：在定义有默认值参数的函数时，只能将默认值赋给最右端的参数，否则会出现语法错误。比如将例 6.5 改成下列程序：

```
def fun(discount=0.5,price):                #定义函数
        print ("商品折扣: ",discount)
        print ("商品价格: ", price)
        return
fun(50)
```

运行将提示错误信息：syntaxError:non-default argument follows default argument

5、不定长参数

在使用函数时，若希望一个函数能够处理比定义时更多的参数，即参数数量是可变的，那么可以在函数中使用不定长参数。与上面几种参数相区别的是，不定长参数在声明时，形参不会命名。在函数的第一行参数列表最右侧增加一个带*的参数，基本语法格式为：

```
def  函数名([形参列表,] *args):
    函数体
```

其中，*args 用来接收任意多个实参并将其放在一个元组(tuple)中供函数使用。

【例 6.7】不定长参数。

程序代码：

```
def f(x, y, *args):
    print(x)
    print(y)
    print(args)
f(10, 9, 8, 7, 6,5,4,3,2,1)
```

运行结果：

```
10
9
(8, 7, 6, 5, 4, 3, 2, 1)
```

6.3.2 函数的返回值

在 Python 中，函数使用 return 语句返回值。选择性的向调用方返回一个表达式。return 语句用来退出函数并将程序返回到函数被调用的位置继续执行，可以返回 0 个，1 个或一组值。函数返回的值被称为返回值。基本语法结构为：

```
return [返回值列表]
```

其中，return 语句可以在函数体的任何位置出现，return 语句是可选的，如果有 return 语句，但是 return 后面没有接表达式或者值的，

则返回 None；如果没有 return 语句，则会自动返回 None。

【例 6.8】编写程序，求某商品的价格。

```
def fun(discount,price):
    price=price*discount
    print ("商品折扣: ",discount)
    print ("商品价格: ", price)
    return price;
price=fun(0.5,100)      #调用 fun 函数
```

运行结果：

```
商品折扣:  0.5
商品价格:  50.0
```

6.4 递归函数

Python 支持函数的递归调用，所谓递归就是函数直接或间接地调用其本身。递归是常用的编程方法，适用于能把一个大型复杂的问题逐层转化为一个与原问题性质相似，但规模较小的问题来求解的场景。

直接递归调用是在调用 f 函数的过程中，又调用 f 函数，间接调用是在调用 f 函数的过程中要调用 f1 函数，在调用 f1 函数的过程中又要调用 f 函数。

【例 6.9】计算 n 的阶乘。

分析：n 的阶乘可以表示为 f(n)=1*2*3…*(n-1)*n，从而可得 f(n)=f(n-1)*n。程序代码如下：

```
def f(n):                       #定义递归函数
    if n==1:                    #当 n 等于 1 时返回 1
        return 1
    else:                       #当 n 不为 1 是返回 f(n-1)*n
        return f(n-1)*n
n = int(input('请输入一个正整数：'))      #输入一个整数
print(n,'的阶乘结果为：',f(n))            #调用函数 f 并输出结果
```

运行结果：

```
请输入一个正整数：3
```

```
3 的阶乘结果为:  6
```

6.5 匿名函数

关键字 lambda 用于定义一种特殊的函数——匿名函数，又称 lambda 函数。匿名函数并非没有名字，而是将函数名作为函数结果返回，这种函数省略了用 def 定义函数的标准步骤。其语法格式如下：

```
函数名 = lambda [参数列表]:表达式
```

简单地说，lambda 函数用于定义简单的、能够在一行内表示的函数，返回一个函数类型。lambda 函数能接收任何数量的参数，但是只能返回一个表达式的值，同时只能处理输出的内容不可包含命令或多个表达式。匿名函数不能直接调用 print,因为 lambda 需要一个表达式。

lambda 函数的语法只包含一个语句，例如：

```
sum=lambda arg1,arg2:arg1+arg2;
print("total:",sum(20,50))   #调用 sum 函数
```

运行结果：

```
total: 70
```

6.6 变量的作用域

扫码看视频 6.3

变量作用域就是变量的使用范围，程序的运行离不开变量。一个程序的所有变量并不是在哪个位置都可以访问，访问权限取决于这个变量是在哪里赋值。

在 python 中，根据变量作用域的不同，分为全局变量和局部变量。

定义在函数内的变量，只能在函数内使用，作用范围仅在函数内部的变量，称为局部变量。在函数之外定义的变量称为全局变量，全局变量在整个程序内起作用。

【例 6.10】变量的作用域

```
price=200                          #全局变量 price
def fun(discount,price):           #定义 fun 函数
    price=price * discount     #局部变量 price
    print ("函数内局部变量价格 : ", price)          #输出局部变量 price 的值
    return price
fun(0.5, 200)                  #调用 fun 函数
print ("函数外全局变量价格 : ", price)        #输出全局变量 price 的值
```

运行结果：

```
函数内局部变量价格:  100.0
函数外全局变量价格:  200
```

在此列中，定义了两个名为 price 的变量，在 fun()函数内部的为局部变量，调用 fun()函数后，函数内输出的值是 100。在函数外，局部变量失效，此时输出函数体外赋予的值 200。

6.7 Python 的内置函数

在 Python 中，系统提供了多种内置函数，也称为内建函数，这些内置函数在使用时不需要引用，可以直接调用。常用的内置函数主要包括数学运算函数，类型转换函数，字符串处理函数及其他函数，如下表：

表 6-1 python常用内置函数汇总

函数名	功能	函数名	功能
abs()	求绝对值	all()	判断参数中的所有数据是否都为True
any()	判断参数中是否存在任意一个为True的数据	complex()	创建一个复数
pow()	幂函数	delattr()	删除对象的属性
bool()	将参数转换成逻辑型数据	dir()	没有参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性和方法列表

chr()	返回对应ASCII码的字符	enumerate()	返回一个可以枚举的对象
bin()	将十进制数转换成二进制数	float()	将参数转换为浮点数
bytes()	将参数转换成字节型数据	frozenset()	创建一个不可修改的集合
hasattr()	判断对象是否具备特定的属性	dict()	创建一个空的字典类型的数据
id()	返回对象的内存地址	divmod()	分别求商和余数
int()	将参数转换成整数	eval()	计算字符串参数中表达式的值
issubclass()	检查一个类是否是另一个类的子类	format()	格式化输出字符串
list()	构造列表数据	getattr()	获取对象的属性
max()	求最大值	hex()	返回参数的十六进制
next()	返回一个可迭代数据结构中的下一项	input()	获取用户输入的内容
open()	打开文件	isinstance()	检查对象是否是类的实例
range()	根据需要生成一个指定的范围	len()	返回对象长度
round()	对参数进行四舍五入	map()	将参数中的所有数据用指定的函数遍历
setattr()	设置对象的属性	min()	返回给定元素中的最小值
str()	构造字符串类型的数据	oct()	将参数转换成八进制
super()	调用父类的方法	ord()	求参数字符的ASCII码
type()	显示对象所属的类型	print()	输出函数
reversed()	反转，逆序对象	sorted()	对参数进行排序
set()	创建一个集合类型的数据	sum()	求和函数
tuple()	构造元组类型的数据	zip()	将两个可迭代对象中的数据逐一配对

下面列举一些内置函数示例：

1.与数学运算相关的内置函数

（1）abs()函数

求取绝对值。函数格式如下：

```
abs(x)
```

abs()函数的参数 a 可以是一个整数、长整数或浮点数。如果参数是复数，则返回它的模。示例：

```
   >>>abs(-5.2)
  5.2
>>>abs(3+4j)
5.0
```

（2）max()函数

求取最大值。函数格式如下：

```
max(x，y，z，…)
```

max()函数的参数可以是多个值，返回其中最大的数值。示例：

```
>>>max(-1,2,4,3)
4
```

（3）min()函数

求取最小值。函数格式如下：

```
min(x，y，z，…)
```

min()函数的参数可以是多个值，返回其中最小的数值。示例：

```
>>>min(-1,2,4,3)
-1
```

（4）pow()函数

获取乘方数。函数格式如下：

```
pow(x,y)
```

这个函数返回 x 的 y 次幂。示例：

```
>>>pow(3,4)
81
```

（5）round()函数

获取指定位数的小数。函数格式如下：

```
round (x,y)
```

其中 x 代表浮点数，y 代表要 保留的位数。示例：

```
>>>round(2.6235,2)
2.62
```

（6）divmod()函数

获取商和余数。函数格式如下：

```
divmod(x,y)
```

示例：

```
>>>divmod(9,2)
 (4, 1)
```

2.类型转换相关的内置函数

类型转换函数是将一种类型的变量强行转化为另一种类型的变量，常见转换函数如下。

（1）int()函数

将参数转换为 int 型。函数格式如下：

```
int(str)
```

示例：

```
>>>int(12.1)
12
```

（2）float()函数

将 int 型或字符型转换为浮点型。函数格式如下：

```
float(int/str)
```

示例：

```
>>>float(3)
3.0
```

（3）str()函数

转换为字符型。函数格式如下：

```
str(int)
```

示例：

```
>>> str(10)
'10'
```

（4）bool() 函数

转换为布尔类型，0 转化为 False，非 0 转化为 True，函数格式如下：

```
bool(int)
```

示例：

```
>>> bool(0)
False
>>> bool(1)
True
```

（5）hex()函数

转换为十六进制。函数格式如下：

```
hex(int)
```

示例：

```
>>> hex(10)
'0xa'
```

（6）bin()函数

转换为二进制。函数格式如下：

```
bin(int)
```

示例：

```
>>> bin(59)
'0b111011'
```

（7）chr()函数

转换数字为相应的 ASCII 码字符。函数格式如下：

```
chr(int)
```

示例：

```
>>> chr(65)
'A'
```

（8）ord() 函数

转换 ASCII 码字符为相应的数字。函数格式如下：

```
ord(str)
```

示例：

```
>>> ord('Z')
90
```

3.字符串中字符大小写变换函数

表6-2 python字符串中字符大小写变换函数表

函数	用法	功能
lower()	str.lower()	将字符串str中的所有字母转换为小写字母
upper()	str.upper()	将字符串str中的所有字母转换为大写字母
swapcase()	str.swapcase()	将字符串str中的所有字母转换为大小写字母互换
capitalize()	str.capitalize()	将字符串str中的首字母大写，其余字母小写
title()	str.title()	将字符串str中的每个单词的首字母大写，其余字母小写

【例 6.11】字符串大小写字母转换

程序代码：

```
str="I am a girl"
print(str.lower())
print(str.upper())
print(str.swapcase())
print(str.capitalize())
print(str.title())
```

运行结果：

```
i am a girl
I AM A GIRL
i AM A GIRL
I am a girl
I Am A Girl
```

4.指定字符串输出方式相关函数

表6-3 字符串输出方式函数

函数	语法	功能
ljust()	str.ljust(width ,[fillchar])	将str左对齐输出，字符串的总宽度为width，不足的部分以fillchar指定的字符串填充，默认使用空格填充
rjust()	str.rjust(width ,[fillchar])	将str右对齐输出，字符串的总宽度为width，不足的部分以fillchar指定的字符串填充，默认使用空格填充
center()	str.center(width ,[fillchar])	将str居中对齐输出，字符串的总宽度为width，不足的部分以fillchar指定的字符串填充，默认使用空格填充
zfill()	str.zfill(width)	将字符串的宽度填充为width，并且右对齐，不

		足的部分用0补足

【例 6.12】指定字符串输出方式

程序代码：

```
str="very good!"
print(str.ljust(16,"-"))
print(str.rjust(16,"-"))
print(str.center(16,"-"))
print(str.capitalize())
print(str.zfill(16))
```

运行结果：

```
very good!------
------very good!
---very good!---
very good!
000000very good!
```

5. 搜索和替换字符串处理函数

表6-4 python搜索和替换字符串处理函数表

函数	语法	功能
find()	str.find(substr[,start,[end]])	从str字符串的start至end的范围内检索是否存在substr，如果存在，则返回出现子串substr的第一个字母的位置，如果str中没有substr，则返回-1
index()	str.index(substr[,start,[end]])	与find()函数相同，只是在str中没有substr时，返回一个运行时的错误
rfind()	str.rfind(substr[,start,[end]])	从str字符串右侧起来的start至end的范围内检索是否存在substr，如果存在，则返回出现子串substr的第一个字母的位置，如果str中没有substr，则返回-1
rindex()	str.rindex(substr[,start,[end]])	与rfind()函数相同，如果没有substr时，返回一个运行时的错误
replace()	str.replace(oldstr,newstr[,count])	把str中的oldstr替换为newstr,count为替换次数

【例 6.13】搜索和替换字符串处理函数示例

```
str="very good!"
```

```
print(str.find('g'))
print(str.index('o'))
print(str.rfind('g'))
print(str.rindex('o'))
print(str.replace("","-"))
```

运行结果：

```
5
6
5
7
-v-e-r-y- -g-o-o-d-!-
```

6.字符串的分割与组合函数

表6-5 分割与组合函数表

函数	语法	功能
split()	str.split([sep,[maxsplit]])	以sep为分隔符，把str分割成一个列表，其中maxsplit表示分割的次数
splitines()	str.splitines(keepends)	把str按照行分割符分为一个列表，其中参数keepends为bool值，当取true时，每行后面会保留行分隔符
join()	str. join(seq)	把seq代表的字符串序列，用str连接起来

【例 6.14】字符串的分割与组合函数示例

程序代码：

```
str = "very good!"
list = str.split('    ')
print(list)
str1 = "-"
print(str1.join(list))
```

运行结果：

```
['very', 'good!']
very-good!
```

7.相关操作函数

（1）type()函数

功能：返回一个对象的类型。type()函数语法：

```
type(x)
```

示例：

```
>>> x=2
>>> type(x)
<class ‘int’>
>>>y=’good’
>>>type(y)
<class ‘str’>
```

(2)help()函数

功能：调用系统内置的帮助。help()函数语法：

```
help(para)
```

示例 1：

```
>>> help('input')
Help on built-in function input in module builtins:
input(prompt=None, /)
    Read a string from standard input.  The trailing newline is stripped.
    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
On *nix systems, readline is used if available.
```

示例 2：

```
>>> help('print')
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

6.8 实验 1：使用函数计算规则图形的面积

#圆形

```
    def findArea(r):
    s=3.14*r*r
    print ("圆的面积: ",s)       #输出局部变量 s 的值
    return s
findArea(2)                 #调用 findArea 函数
```

#长方形

```
def findArea(w,h):
    s=w*h
    print ("长方形的面积: ",s)
    return s
findArea(2,3)
```

6.9 实验 2：利用递归函数调用方式，将所输入的 5 个字符，以相反顺序打印出来

```
def func(s):
        try:
            print(s[-1])
            func(s[0:len(s)-1])
        except Exception:
            pass
func('12345')
```

运行结果：

```
5
4
3
2
1
```

第七单元：正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。典型的搜索和替换操作要求提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

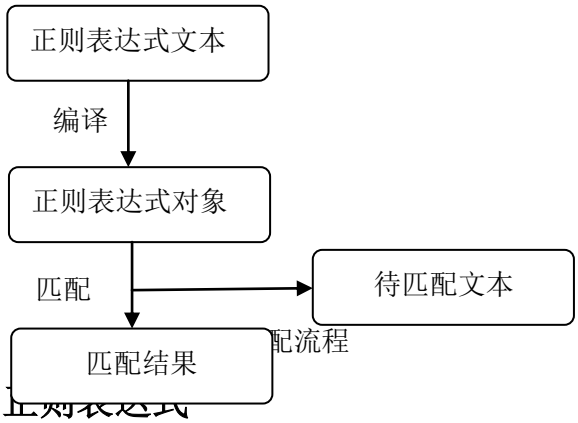
7.1 概述

正则表达式：又称规则表达式，实际上是一个“文本模式”，通过一个字符序列来定义一种搜索模式，主要用于字符串模式匹配或字符串匹配（即查找和替换操作）。一个正则表达式由字母、数字和一些特殊符号组成，这些符号的序列组成一个“规则字符串”，用来表示满足某种逻辑条件的字符串。

给定一个正则表达式和一个普通字符串，我们可以：

判断普通字符串（或其子串）是否符合正则表达式所定义的逻辑（字符串与正则表达式是否匹配）；从字符串中提取或替换某些特定部分。

许多编程语言都对正则表达式提供了不同程度的支持，图 7-1 展示了使用正则表达式进行匹配的流程。



配流程

7.2 为何需要使用正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。典型的搜索和替换操作要求提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

通过使用正则表达式，可以：

1、测试字符串内的模式。

例如，可以测试输入字符串，查看字符串内是否出现电话号码模式或信用卡号码模式。这称为数据验证。

2、替换文本。

可以使用正则表达式来识别文档中的特定文本，完全删除该文本或者用其他文本替换它。

3、基于模式匹配从字符串中提取子字符串。

4、可以查找文档内或输入域内特定的文本。

7.3 正则表达式的语法

正则表达式(regular expression)由普通字符（例如字符 a 到 z）和一些特殊字符（元字符 metacharacters）组成的文本模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模式，将某个字符模式与所搜索的字符串进行匹配。元字符是正则表达式中具有特殊含义的字符，用来匹配一个或者若干个满足某种条件的字符。这些元字符是构成正则表达式的关键要素。

构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。下面分类列出最常用的元字符及其含义。

7.3.1 python 中的字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印

字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

1.非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

表7-1 非打印字符表

字符	描述
\cx	匹配由x指明的控制字符。例如： \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

2．特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如字符串 zunoo*b 中的 *，简单的说就是表示任何字符串的意思。如果要查找字符串中的 * 符号，则需要对 * 进行转义，即在其前加一个 \: zuno*ob 匹配 zuno*ob。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符\ 放在它们前面。下表列出了正则表达式中的特殊字符：

表7-2 正则表达式中的特殊字符

特殊字符	描述

$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*****	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
****	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
|	指明两项之间的一个选择。要匹配 |，请使用 \\|。

3．数量限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 * 或 + 或 ? 或 {n} 或 {n,} 或 {n,m} 共 6 种。正则表达式的限定符有：

表7-3 正则表达式的限定符

字符	描述
*****	匹配前面的子表达式零次或多次。例如，go* 能匹配 "g" ，”go”以及 "goo"。* 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，go+' 能匹配 "go" 以及 "goo"，但不能匹配 "g"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"go?" 可以匹配 "g"以及"go",但不能匹配”goo”。? 等价于 {0,1}。
{n}	n是一个非负整数。用来匹配前面的字符串 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n是一个非负整数。至少匹配n次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹

	配 "foooood" 中的所有o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m和n 均为非负整数，其中n <= m。最少匹配n次且最多匹配m次。例如，"ab{1,3}" 能匹配“ab”或“abbb”,但不能匹配“abbbb”。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

4．定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。定位符用来描述字符串或单词的边界，^ 和 $ 分别指字符串的开始与结束，\b 描述单词的前或后边界，\B 表示非单词边界。

表 7-5 正则表达式的定位符

字符	描述
^	匹配输入字符串开始的位置。
$	匹配输入字符串结尾的位置。
\b	匹配一个单词边界，即字与空格间的位置。
\B	非单词边界匹配。

5. 正则表达式特殊序列

字符	描述
\d	用来匹配一个数字字符，相当于[0-9]
\D	用来对数位类求反，相当于[^0-9]
\w	用来匹配一个单词字符，包括字母、数字和下划线，相当于[a-zA-Z0-9]
\W	对字母、数字和下划线求反，相当于[^a-zA-Z0-9]
\s	用来匹配一个不可见字符（包括空格、制表符和换行符），相当于[\n\f\r\t\v]
\S	用来匹配一个可见字符[^\n\f\r\t\v]

接下来将演示一个包含类和特殊特殊字符序列的正则表达式。

【例 7.1】使用类和特殊序列的正则表达式。

程序代码：

```
import re     #导入 re 模块
#用[]指定字符类
```

```
testStrings=["3x+6y","5y-4z"]
expressions=[r"3x\+6y|5y-4z",r"[0-9][a-zA-z0-9_].[0-9][yz]",r"\d\w-\d\w"]
#每一个表达式和每一个字符串相匹配
for expression in expressions:
    for string in testStrings:
        if re.match(expression,string):
            print(expression,"matches",string)
#指定字符类和特殊序列
testString1="010-123-4567"
testString2="871-123-4567"
testString3="email:\t joe @deitel.com"
expression1=r"^\d{3}-\d{3}-\d{4}$"
expression2=r"\w+:\s+\w+@\w+.(com|org|net)"
#匹配字符类
if re.match(expression1,testString1):
    print(expression1,"matches",testString1)
if re.match(expression1,testString2):
    print(expression1,"matches",testString2)
if re.match(expression1,testString1):
    print(expression2,"matches",testString3)
```

运行结果：

```
3x\+6y|5y-4z matches 3x+6y
3x\+6y|5y-4z matches 5y-4z
[0-9][a-zA-z0-9_].[0-9][yz] matches 3x+6y
[0-9][a-zA-z0-9_].[0-9][yz] matches 5y-4z
\d\w-\d\w matches 5y-4z
^\d{3}-\d{3}-\d{4}$ matches 010-123-4567
^\d{3}-\d{3}-\d{4}$ matches 871-123-4567
\w+:\s+\w+@\w+.(com|org|net) matches email:      joe@deitel.com
```

7.4 在 Python 中使用正则表达式

扫码看视频 7.1

Python 自 1.5 版本起增加了 re 模块，re 模块使 Python 语言拥有全部的正则表达式功能。

re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

Python 在标准库中提供了 re 模块来处理正则表达式，re 模块提供了一些函数，接下来介绍 re 模块中的 compile()函数、match()函数、

search()函数对正则表达式进行编译、匹配字符串、查找，替换等操作。

7.4.1 compile()函数

compile()函数用来对正则表达式进行编译和处理正则表达式对象，它的格式如下：

```
re.compile(pattern,flags=0)
```

其中，flags 是可选的编译标志，返回一个 pattern 对象。re 模块函数几乎都可以用 pattern 对象的方法，还可以将正则表达式编译成特定形式，以便模块将其用于处理一个字符串。下面将用实例来说明如何提前编译正则表达式，创建表达式的对象，并利用 re.search 返回对象查看搜索结果。

【例 7.2】使用 compile()函数编译正则表达式

```
import re        #导入 re 模块
testString="very good"     #定义一个字符串
formatString="%-35s:%s"
expression="very"       #创建正则表达式
compiledExpression=re.compile(expression)        #编译正则表达式
print(formatString % ("The expression",expression))       #输出正则表达式
print(formatString % ("The compiled expression",compiledExpression))
print(formatString % ("non-compiled search",re.search(expression,testString)))
print(formatString % ("compiled search",compiledExpression.search(testString)))
print(formatString%("search SRE_Match contains",
re.search(expression,testString).group()))
print(formatString % ("compiled search SRE_Match contains",
compiledExpression.search(testString).group()))
```

运行结果：

```
The expression :very
The compiled expression:re.compile('very')
ono-compiled search:<re.Match object; span=(0, 4), match='very'>
compiled search:<re.Match object; span=(0, 4), match='very'>
search SRE_Match contains:very
compiled search SRE_Match contains :very
```

程序说明，第 1 行导入正则表达式 re 模块，创建编译正则表达式“very good”,第 5 行的 re.compile 函数取得一个正则表达式作为参数，并返回一个 SRE_pattern 对象，它代表编译好的正则表达式。编译好的正则表达式对象具有 re 模块的所有功能。例如，在第 9 行中 ompiledExpression 的 search 方法对应于 re.search 函数。输出结果显示，这两种方式都能返回一个 SRE_Match 对象。在第 10 行，11 行中利用 group 方法返回匹配的字符串。

7.4.2 match()函数

match()函数用来匹配对象。它的格式如下：

```
re.match(pattern,string,flags=0)
```

match()函数将会从字符串 string 开头扫描若干字符是否匹配正则表达式 pattern，如果匹配成功则返回匹配对象，否则返回 None。下面将用实例来说明使用 match()来匹配字符串的过程，在这个例子中，我们匹配单个字符，例如，在第 3 行中的表达式“hel?o”，它的匹配模式由一个字母 h，字母 e，0 个或多个字母 l 和字母 o 组成。正则表达式"hel+o"，它的匹配模式由一个字母 h，字母 e，1 个或多个字母 l 和字母 o 组成。正则表达式"hel*o"，它的匹配模式由一个字母 h，字母 e，0 个或多个字母 l 和字母 o 组成。

【例 7.3】使用 match()函数匹配字符串

程序代码：

```
import re
testStrings=["heo","helo","hellllo"]
expressions=["hel?o","hel+o","hel*o"]
for expression in expressions:
    for string in testStrings:
```

```
            if re.match(expression,string):
                print(expression,"matches",string)
            else:
                print(expression,"does not match",string)
                print
    expression1="elo"
    expression2="^elo"
    expression3="elo$"
    if re.match(expression1,testStrings[1]):
        print(expression1,"matches",testStrings[1])

    if re.match(expression1,testStrings[1]):
        print(expression1,"found in",testStrings[1])

    if re.match(expression2,testStrings[1]):
        print(expression2,"found in",testStrings[1])

    if re.match(expression3,testStrings[1]):
        print(expression3,"found in",testStrings[1])
```

运行结果：

```
hel?o matches heo
hel?o matches helo
hel?o does not match hellllo
hel+o matches helo
hel+o matches hellllo
hel+o does not match hellllo
hel*o matches heo
hel*o matches helo
hel*o matches hellllo
hel*o does not match hellllo
```

程序说明，第 1 行导入正则表达式 re 模块，第 3 行创建一个正则表达式列表，该列表中包含元字符？、+、*，元字符的使用（参见第 7.3.1 节）。第 4 行至 10 行使用一个嵌套 for 循环，它将第 3 行的每个正则表达式应用于第 2 行的每一个字符串。第 14 行中使用 re.match 函数将一个表达式与一个字符串匹配。正则表达式中还包含了 2 个元字^和$，分别位于字符串的开头和末尾。在进行搜索和匹配时，只有在字符串开头和末尾包含指定模式的前提下，才会返回一个

值，第 11 行至 21 行创建包含这些元字符的正则表达式，并用函数 re.search 来搜索元字符串，用 re.match 函数来匹配字符串。

7.4.3 search()函数


扫码看视频 7.2

使用 search()函数来搜索字符串，它的格式如下：

```
re.search(pattern,string,flags=0)
```

这个函数将扫描字符串 string，找到第一个与正则表达式 pattern 匹配的位置，并返回对应的匹配对象，如果不存在一个匹配，则返回 None。下面将用实例来说明使用 search()函数来查找字符串的过程。在本例中查找的字符串是：”very”，”Very”和”good”

【例 7.4】使用 search()函数查找字符串

程序代码：

```
import re          #导入 re 模块
testStrings=["Very good","very good!","very good"]   #定义字符串列表
expressions=["very","Very","good!"]      #定义正则表达式搜索的字符串列表
for string in testStrings:   #遍历字符串
    for expression in expressions:
        if re.search(expression,string):   #判断是否搜索到匹配的字符串
            print(expression,"found in string",string)   #若匹配，输出匹配的字符串
        else:
                print(expression,"not found in string",string)
                print
```

运行结果：

```
very not found in string Very good
Very found in string Very good
good! not found in string Very good
very found in string very good!
Very not found in string very good!
good! found in string very good!
very found in string very good
Very not found in string very good
good! not found in string very good
```

程序说明，第 1 行导入正则表达式 re 模块，它为 python 提供了正则表达式处理能力。第 2 行 testStrings 列表，列出了包含几个要用第 3 行创建正则表达式搜索的字符串。在第 4 行到 10 行使用 for 循环嵌套，目的是检测列表 testStrings 中的每一个字符串，判断其中是否包含列表 expressions 中的任何一个正则表达式。第 6 行使用 if 条件函数来判断 re.search 函数是否搜索到指定的字符串，如果搜索到，则返回一个对象，该对象中包含与正则表达式匹配的子字符串。若搜索不到指定字符串，则返回 None。程序确定函数调用是否返回一个值，然后输出相应的消息。

7.4.4 sub()函数和 split()函数

扫码看视频 7.3

sub()函数主要用于搜索和替换。它的格式如下：

```
re.sub(pattern,rep1,string,flags=0)
```

其中，pattern 是编译前的正则表达式字符串，rep1 可以是一个单参数的函数，接收匹配串作为参数并返回新串以替换。String 是进行匹配的字符串，flags 是正则表达式匹配时的选项。该函数用来将字符串 string 对正则表达式 pattern 的每个非重叠匹配项替换为字字符串 rep1，并返回替换后的新串。

Split()函数用于将字符串 string 以正则表达式 pattern 的匹配项分隔符进行拆分，并返回拆分后的字符串列表。可选参数 maxsplit 大于 0 时表示最大的拆分数量。例如：调用 re.split(r",",A,B,C)，将返回字符串列表[‘A’,’B’,’C’]其格式为：

```
re.split(pattern,string,maxsplit=0,flags=0)
```

【例 7.5】编程实现将某个字符串替换成子字符串

程序代码：

```
import re
testString1="This sentence ends in 5 stars *****"
testString2="1,2,3,4,5,6,7"
testString3="1+2x*3-y"
formatString="%-34s:%s"
print(formatString % ("Original string",testString1))
testString1=re.sub(r"\*", r"^", testString1)
print(formatString % ("^substituted for *", testString1))
testString1=re.sub(r"stars", "carets", testString1)
print(formatString %('"carets" substituted for "starts"',testString1))
print(formatString % ('Every word replaced by "word"',
re.sub(r"\w+","word", testString1)))
print(formatString %('Replace first 3 digits by "digit"',
re.sub(r"\d","digit", testString2,3)))
print(formatString %("Splitting"+testString2,re.split(r",",testString2)))
print(formatString %("Splitting"+testString3,re.split(r"[+\-*/%]",testString3)))
```

运行结果：

```
Original string:This sentence ends in 5 stars *****
^substituted for *:This sentence ends in 5 stars ^^^^^
"carets" substituted for "starts" :This sentence ends in 5 carets ^^^^^
Every word replaced by "word":word word word word word word ^^^^^
Replace first 3 digits by "digit" :digit,digit,digit,4,5,6,7
Splitting1,2,3,4,5,6,7:['1', '2', '3', '4', '5', '6', '7']
Splitting1+2x*3-y:['1', '2x', '3', 'y']
```

程序说明，本例使用正则表达式来对字符串进行处理，第 1 行导入正则表达式 re 模块，第 2，3,4 行定义三个字符串，第 5 行定义字符串的输出格式，第 6 行输出原始字符串 testString1，第 7 行使用 re.sub 函数取得三个参数，第一个参数指定模式，第二个参数指定子字符串，第三个参数指定字符串。在 3 个参数指定的字符串中，与模式匹配的每个字符串都会被替换为第二个参数指定的字符串中。第 7 行中用”^”替换字符串 testString1 中的“*”。第 11 行，将 testString1 中每一个单词都替换成字符串”word”。第 12 行中，使用 re.sub 函数可选的第 4 个参数指定最多替换次数。在 13，14 行中，调用 re.split

函数对字符串 testString2 和 testString3 中的参数进行分解，返回一个字符列表。

7.5 捕获

捕获和分组在正则表达式中有着密切的联系，一般情况下，分组即捕获，用一对圆括号“()”括起来的正则表达式，匹配出的内容就表示一个分组，举个例子，假设我们需要匹配一个座机号码：

```
 m = re.search(r'^(\d{3,4}-)?(\d{7,8})$','010-62226666)
m.group(0)
'010-62226666'
m.group(1)
'010-'
m.group(2)
'62226666'
```

这里，默认分组 group(0)是完整的匹配，之后的分组则按出现顺序排列，接下来，如果我们想在一整段文本中，找出所有的座机号码，这里需要用到 re.findall:

```
re.findall(r'(\d{3,4}-)?(\d{7,8})','010-62226666\n0357-4227865')
[('010-62226666'), ('0357-', '4227865')]
```

findall 有一个特性，就是如果结果中有捕获的分组，则将捕获的分组组成 tuple 返回。利用这个特点，和上面提到的分组，但是不捕获的语法，可以得到我们想要的结果：

```
re.findall(r'(?:\d{3,4}-)?\d{7,8}','010-62226666\n0357-4227865')
['010-62226666', '0357-4227865']
re.findall(r'(?:\d{3,4}-)?\d{7,8}','010-62226666\n4227865')
['010-62226666', '4227865']
```

7.6 贪婪与非贪婪模式

所谓“贪婪”，其实就是在多种长度的匹配字符串中，选择较长的那一个。默认情况下，正则表达式将进行贪婪匹配。

在 python 中，元字符*、+限定符都是贪婪的，因为它们会尽可

能多的匹配文字，只有在它们的后面加上一个?就可以实现非贪婪或最小匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

<H1>正则表达式的使用</H1>

贪婪：表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

非贪婪：如果您只需要匹配开始和结束 H1 标签，下面的非贪婪表达式只匹配 <H1>。

```
/<.*?>/
```

如果只想匹配开始的 H1 标签，表达式则是：

```
/<\w+?>/
```

通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从“贪心”表达式转换为“非贪心”表达式或者最小匹配。

【例 7.6】正则表达式的贪婪与非贪婪运算符

```
import re
formatString1="%-22s:%s"
testString1=\
                    "Albert Antstein,phone:123-4567,e-mail:albert@bug2bug.com"
expression1=\
                    r"(\w+ w+),phone:(\d{3}-\d{4}),e-mail:(\w+@w+\.\w{3})"
print(formatString1 % ("Extract all user data",
re.match(expression1,testString1).group()))
print(formatString1 % ("Extract user e-mail",
re.match(expression1,testString1).group(3)))
print
formatString2="%-38s:%s"
pathstring="/books/2019/python"
expression2="(/.+)/"
print(formatString1%("greedy error",re.match(expression2,pathstring).group(1))
```

```
expression3="(/.+?)/"
print(formatString1 % ("non error,base only",
re.match(expression3,pathstring).group(1)))
```

运行结果：

```
Extract all user data：（'Albert Antstein',' 123-4567',' albert@bug2bug.com'）
Extract user e-mail: albert@bug2bug.com
greedy error: /books/2019
non error,base only: /books
```

程序说明，此例中创建了含有分组的正则表达式，从这些分组中获取信息，第 6 行中的正则表达式描述了这些组，使用了 python 中的元字符，第一组匹配的模式是一个单词（\W+）,第二组匹配 3 个数位，后跟-，最后是 4 个数位。第 3 组匹配一个或多个字符， 后跟@符号。这个正则表达式与 testString1 中的字符进行匹配。第 7 至 10 行进行分组，第 8 行调用 re.match() 函数，、第 8 行是获得这个人的姓名，号码，电子邮件地址。

7.7 典型案例

用户账号密码登录验证

当登录一个网站时，经常输入用户名、密码等信息，而网站一般对这些信息的长度和格式都有要求。当输入不符合格式的信息时，网站会自动提示输入信息格式有误。给出一个使用正则表达式校验信息格式的示例。

程序代码：

```
import re
def user_name(name, password):
    # "判断用户名密码是否合法！"
      result_name = re.compile(r"[\u4e00-\u9fa5]")
      result_password = re.compile(r"^[a-zA-Z]\w{6,18}")

      if result_name.match(name) and    result_password.match(password):
```

```
            print("登录成功")
        else:
            print("用户名不合法请您重新输入")

def main():
    #   "main  总入口函数"
        name = input("请输入你的用户名：")
        password = input("请输入你的密码：")
        user_name(name, password)
if __name__ == '__main__':
        main()
```

运行结果：

```
请输入你的用户名：1234567
请输入你的密码：2222
用户名不合法请您重新输入
请输入你的用户名：abc_123
请输入你的密码：pytH787878
登录成功
```

第八单元：面向对象编程

8.1 面向对象编程概述

面向对象程序设计（Object Oriented Programming，OOP）的思想主要针对大型软件设计而提出，使得软件设计更加灵活，能够很好地支持代码复用和设计复用，并且使得代码具有更好的可读性和可扩展性。面向对象程序设计的一条基本原则是计算机程序由多个能够起到子程序作用的单元或对象组合而成，这大大地降低了软件开发的难度，使得编程就像搭积木一样简单。面向对象程序设计的一个关键性观念是将数据以及对数据的操作封装在一起，组成一个相互依存、不可分割的整体，即对象。对于相同类型的对象进行分类、抽象后，得出共同的特征而形成了类，面向对象程序设计的关键就是如何合理地定义

和组织这些类以及类之间的关系。

Python 完全采用了面向对象程序设计的思想，是真正面向对象的高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。但与其他面向对象程序设计语言不同的是，Python 中对象的概念很广泛，Python 中的一切内容都可以称为对象，而不一定必须是某个类的实例。例如，字符串、列表、字典、元组等内置数据类型都具有和类完全相似的语法和用法。创建类时用变量形式表示的对象属性称为数据成员或成员属性，用函数形式表示的对象行为称为成员函数或成员方法，成员属性和成员方法统称为类的成员。

8.1.1 面向对象的基本概念

1．对象

现实世界中客观存在的事物称作对象（object），任何对象都具有各自的特征（属性）和行为（方法）。

面向对象程序设计中的对象是现实世界中的客观事物在程序设计中的具体体现，它也具有自己的特征和行为。对象的特征用数据来表示，称为属性（property）。对象的行为用程序代码来实现，称为对象的方法（method）。总之，任何对象都是由属性和方法组成的。

2．类

类（class）是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述。任何对象都是某个类的实例（instance）。

在系统中通常有很多相似的对象，它们具有相同名称和类型的属性、响应相同的消息、使用相同的方法。将相似的对象分组形成一个类，每个这样的对象被称为类的一个实例，一个类中的所有对象共享一个公共的定义，尽管它们对属性所赋予的值不同。

3．消息

一个系统由若干个对象组成，各个对象之间通过消息（message）相互联系、相互作用。消息是一个对象要求另一个对象实施某项操作的请求。发送者发送消息，在一条消息中，需要包含消息的接收者和要求接收者执行某项操作的请求，接收者通过调用相应的方法响应消息，这个过程被不断地重复，从而驱动整个程序的运行。

4．封装

封装（encapsulation）是指把对象的数据（属性）和操作数据的过程（方法）结合在一起，构成独立的单元，它的内部信息对外界是隐蔽的，不允许外界直接存取对象的属性，只能通过使用类提供的外部接口对该对象实施各项操作，保证了程序中数据的安全性。

类是数据封装的工具，对象是封装的实现。类的访问控制机制体现在类的成员中可以有公有成员、私有成员和保护成员。对于外界而言，只需要知道对象所表现的外部行为，而不必了解内部实现细节。

5．继承

继承（inheritance）反映的是类与类之间抽象级别的不同，根据继承与被继承的关系，可分为基类和衍类，基类也称为父类，衍类也称为子类，正如“继承”这个词的字面含义一样，子类将从父类那里获

得所有的属性和方法，并且可以对这些获得的属性和方法加以改造，使之具有自己的特点。

6．多态

多态（polymorphism）是指同一名字的方法产生了多个不同的动作行为，也就是不同的对象收到相同的消息时产生不同的行为方式。

将多态的概念应用于面向对象程序设计，增强了程序对客观世界的模拟性，使得对象程序具有了更好的可读性，更易于理解，而且显著提高了软件的可复用性和可扩充性。

8.1.2 从面向过程到面向对象

面向过程程序设计就是采用自顶向下的方法，分析出解决问题所需要的步骤，将程序分解为若干个功能模块，每个功能模块用函数来实现。

一个面向对象的程序一般由类的声明和类的使用两部分组成。程序设计始终围绕“类”展开。通过声明类，构建了程序所要完成的功能，体现了面向对象程序设计的思想。在 Python 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类的概念。

8.2 创建类与对象


扫码看视频 8.1

类是面向对象程序设计的基础，可以定义指定类的对象。类中可以定义属性（特性）和方法（行为）。

8.2.1 定义类

在 Python 中，通过 class 关键字来定义类。定义类的语法格式如

下：

```
class 类名:
成员变量
成员函数
```

例如定义了一个 Person 类：

```
class Person:
    name='brenden'                    #定义了一个属性
    def printName(self):              #定义了一个方法
        print(self.name)
```

8.2.2 对象的创建和使用

在 Python 中，用赋值的方式创建类的实例，一般格式为：

```
对象名=类名(参数列表)
```

创建对象后，可以使用“.”运算符，通过实例对象来访问这个类的属性和方法（函数），一般格式为：

```
对象名.属性名
对象名.函数名()
```

例如,语句“p=Person()”产生了一个 Person 的实例对象，此时可以用 p.name 来调用类的 name 属性。

【例 8.1】类和对象应用示例。程序如下：

```
class CC:
    x=10                    #定义属性
    y=20                    #定义属性
    z=30                    #定义属性
    def show(self):             #定义方法
        print((self.x+self.y+self.z)/3)
b=CC()          #创建实例对象 b
b.x=30          #调用属性 x
b.show()        #调用方法 show
```

运行结果：

```
26.666666666666668
```

8.3 属性和方法

8.3.1 类属性和实例属性

1．类属性

类属性被所有类对象的实例对象所共有。类属性通常在类体中初始化。对于公有的类属性，在类外可以通过类对象和实例对象访问。例如：


扫码看视频 8.2

【例 8.2】定义 Student 类，定义实例属性和方法

```
class Person:
    name='brenden'        #公有的类属性
    __age=18               #私有的类属性
p=Person()
print(p.name)              #正确，但不提倡
print(Person.name)         #正确
print(p.__age)             #错误，不能在类外通过实例对象访问私有的类属性
print(Person.__age)        #错误，不能在类外通过类对象访问私有的类属性
```

2．实例属性

实例属性（instance attribute)是不需要在类中显式定义，而在__init__构造函数中定义的，定义时以“self. ”作为前缀，实例属性属于特定的实例。在其他方法中也可以随意添加新的实例属性,但并不提倡这么做，所有的实例属性最好在__init__中给出。实例属性在内部通过“self. ”访问，在外部通过对象实例访问。例如：

【例 8.3】定义 Student 类，定义实例属性和方法

```
class Student:
    def __init__(self, name, age, grade):
        self.name=name
        self.age=age
        self.grade=grade
    def say_hi(self):
        print('I am a student, my name is', self.name)

s1=Student('王子', 21, 3)
s1. say_hi()
```

```
print(s1. grade)
s2= Student('公主', 20, 2)
s2. say_hi()
print(s2. grade)
```

运行结果：

```
I am a student, my name is 王子
3
I am a student, my name is 公主
2
```

上述例子中，Student 类中定义了实例属性 name、age 和 grade。s1、s2 是 Student 的两个例，这两个实例分别拥有自己的属性值，有别于其他对象。

实例的属性可以像普通变量一样进行操作，例如

```
s1.grade=s1.grade+1
```

8.3.2 类的方法

方法是与类相关的函数。方法和函数不同，函数是封装操作的小程序。方法是定义在类内部的函数，并且定义方法与普通函数有所不同。

类的方法主要有三种类型：实例方法、类方法和静态方法，不同的方法有不同的使用场景和声明调用形式，不同的方法也具有不同的访问限制。实例方法是属于实例的方法，通过实例名方法名调用，该方法可以访问类属性、实例属性、类方法、实例方法和静态方法。类方法是属于类的方法，可以通过实例名方法名，也可以通过类名方法名调用。类方法不能访问实例属性和实例方法。静态方法是与类实例对象无关的方法，调用形式同类方法类似。

1.方法的声明和调用

在类的内部，使用 def 关键字可以为类定义一个方法。与一般函

数定义不同，类方法必须包含对象本身的参数，通常为 self，且为第一个参数。声明方法的语法格式如下：

def 方法名(self,[形参列表])

函数体

方法调用的语法格式如下：

对象.方法名([参数列表])

在调用方法时，第一个参数 self 不需要用户为其赋值，Python 自动把对象实例传递给参数 self。

2．实例方法

实例方法是在类中最常定义的成员方法，它至少有一个参数并且必须以实例对象作为其第一个参数，一般以“self”作为这第一个参数。在类外，实例方法只能通过实例对象去调用。实例方法的声明格式如下：

def 方法名(self，[形参列表]):

函数体

实例方法通过实例对象调用：

对象.方法名([实参列表])

【例 8.4】定义 Person 类，通过实例调用实例方法

```
class Person:
    place='Changsha'
    def getPlace(self):          #实例方法
        return self.place
p=Person()
print(p.getPlace())             #正确，可以用过实例对象引用
print(p.place)
```

运行结果：

```
Changsha
Changsha
```

3.类方法

类方法不对特定实例进行操作，在类方法中访问实例属性会导致错误。类方法需要用修饰器“@ classmethod”来标识其为类方法。对于类方法，第一个参数必须是类对象，一般以“cls”作为这第一个参数，类方法可通过实例对象和类对象去访问。类方法的声明格式如下：

@classmethod

def 类方法名(cls，[形参列表]):

函数体

类方法调用格式如下：

类名.类方法([实参列表])

通过实例对象和类对象去访问类方法。

【例 8.5】定义 Person 类，定义并调用类方法

```
class Person:
    place='Changsha'
    @classmethod          #类方法，用@classmethod 来进行修饰
    def getPlace(cls):
        return cls.place
p=Person()
print(p.getPlace())          #可以用过实例对象引用
print(Person.getPlace())     #可以通过类对象引用
```

运行结果：

```
Changsha
Changsha
```

类方法还有一个用途就是可以对类属性进行修改。

【例 8.6】定义 Person 类，定义并修改类属性

```
class Person:
    place='Changsha'
```

```
    @classmethod
    def getPlace(cls):
        return cls.place
    @classmethod
    def setPlace(cls,place1):
        cls.place=place1
p=Person()
p.setPlace('Shanghai')          #修改类属性
print(p.getPlace())
print(Person.getPlace())
```

运行结果：

```
Shanghai
Shanghai
```

4．静态方法

静态方法需要通过修饰器“@staticmethod”来进行修饰，静态方法不需要多定义参数。

【例 8.7】定义 Person 类，定义并调用静态方法

```
class Person:
    place='Changsha'
    @staticmethod
    def getPlace():             #静态方法
        return Person.place
print(Person.getPlace())
```

运行结果：

```
Changsha
```

8.3.3 构造方法和析构方法

类中最常用的内置方法就是构造方法和析构方法。

1．构造方法

构造方法__init__(self,……)在生成对象时调用，可以用来进行一些属性初始化操作，不需要显式去调用，系统会默认去执行。构造方法支持重载，如果用户自己没有重新定义构造方法，系统就自动执行默认的构造方法。

【例 8.8】构造方法使用示例

```
class Person:
    def __init__(self,name):
        self.PersonName=name
    def sayHi(self):
        print('大家好，我是{}。'.format(self.PersonName))
p=Person('王子')
p.sayHi()
```

运行结果：

```
大家好，我是王子。
```

2．析构方法

析构方法__del__(self)在释放对象时调用，支持重载，可以在其中进行一些释放资源的操作，不需要显式调用。下面的例子说明了类的普通成员函数以及构造方法和析构方法的作用。

【例 8.9】析构方法示例

```
class Test:
    def __init__(self):
        print('构造方法')
    def __del__(self):
        print('析构方法')
    def myf(self):
        print('调用自定义方法')
obj=Test()
obj.myf()
del obj
```

运行结果：

```
构造方法
调用自定义方法
析构方法
```

8.3.4 属性和方法的访问控制

1．属性的访问控制

在 Python 中没有像 C++中 public 和 private 这些关键字来区别公有属性和私有属性，它是以属性命名方式来区分，如果在属性名前面

加了两个下划线“__”，则表明该属性是私有属性，否则为公有属性。方法也一样，如果在方法名前面加了 2 个下划线，则表示该方法是私有的，否则为公有的。例如：

【例 8.10】定义 Person 类，调用其公有属性

```
class Person:
    name='王子'
    age=18
p=Person()
print(p.name,p.age)
```

运行结果：

```
王子 18
```

这里的 name 和 age 都是公有的，可以直接在类外通过对象名访问，如果想定义成私有的，则需在前面加 2 个下划线“__”。

2．方法的访问控制

在类中可以根据需要定义一些方法，定义方法采用 def 关键字，在类中定义的方法至少会有一个参数，一般以名为“self”的变量作为该参数（用其他名称也可以），而且需要作为第一个参数。下面看一个例子。

【例 8.11】方法的访问控制使用示例

```
class Person:
    __name='公主'
    __age=16
    def getName(self):
        return self.__name
    def getAge(self):
        return self.__age
p=Person()
print(p.getName(),p.getAge())
```

运行结果：

```
公主 16
```

__name 和__age 是类的私有属性，getName 和 getAge 是类的方法。紧接着定义了实例对象 p。如果想获得姓名年龄的信息，不能使用 p. __name 和 p. __age 直接获取，必须要使用 p.getName()和 p.getAge()才能获取。

8.4 继承


扫码看视频 8.3

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承用于指定一个类将从其父类获取其大部分或全部功能。用户对现有类进行修改，来创建一个新的类，称为子类或派生类，原有的类称为基类或父类。它是面向对象编程的一个特征。

子类继承父类的功能，向其添加新功能。它有助于代码的可重用性。继承的语法格式如下：

class 子类名(父类名):

类体

在定义一个类的时候，可以在类名后面紧跟一对括号，在括号中指定所继承的父类，如果有多个父类，多个父类名之间用逗号隔开。

【例 8.12】面向对象之继承示例

```
#例 8.12 面向对象之继承
class Person (object):
    def __init__(self, name, gender):
        self.name= name
        self.gender= gender
        print(" Person 类__ini()__。","姓名：",self.name)
class Student(Person):
    def __init__ (self, name, gender, score):
```

```
        super (Student,self).__init__ (name, gender)
        self.score= score
        print(" Student 类__ini__()。","姓名：  ",self.name)
#“__main__”等于当前执行文件的名称
if __name__=="__main__":
    person= Person("张三","男")
    student= Student("李四","男",100)
```

运行结果：

```
Person 类__ini()__。  姓名：  张三
Person 类__ini()__。  姓名：  李四
Student 类__ini__()。  姓名：  李四
```

继承的一个弱点就是，可能特殊的本类又有其他特殊的地方，又会定义一个类，其下也可能再定义类，这样就会造成继承的那条线越来越长，使用继承的话，任何一点小的变化也需要重新定义一个类，很容易引起类的爆炸式增长，产生一大堆有着细微不同的子类。

子类可以继承一个基类，也可以继承多个基类，这就是多重继承。多重继承的语法格式如下：

class 子类名(父类名 1,父类名 2,……):

类体

【例 8.13】面向对象之多重继承

```
#例 8.13 面向对象之多重继承
class P1():
    def foo(self):
        print("pl-foo")
class P2():
    def foo(self):
        print("p2-foo")
    def bar(self):
        print("p2-bar")
class C1(P1,P2):
    pass
class C2(P1,P2):
    def bar(self):
        print ("C2-bar")
```

```
class D(C1,C2):
    pass
if __name__=='__main__':
    d=D()
    d.foo()
    d.bar()
```

运行结果：

```
p1-foo
C2-bar
```

8.5 多态


扫码看视频 8.4

面向对象程序设计语言有三大特性：继承、多态和封装。Python 语言作为一门面向对象编程语言也不例外，除去前面介绍的继承特性外，也具有多态和封装特性。多态意味着不同类的对象使用相同的操作。封装意味着对外部隐藏对象的行为。

多态即多种形态，在运行时确定其状态，在编译阶段无法确定其类型，这就是多态。例如，序列类型有多种形态：字符串、列表、元组。多态性指的是：向不同对象发送同一条消息，不同对象在接收时会产生不同的行为(即方法)。所谓消息，就是调用函数，不同的行为就是指不同的实现，即执行不同的函数。

在继承关系中，派生类覆盖父类的同名方法，当调用同名方法的时候，系统会根据对象来判断执行哪个方法，这就是多态性的体现。

len 函数不仅可以计算字符串的长度，还可以计算列表、元组等对象中的数据个数，这里在运行时通过参数类型确定其具体的计算过程，正是多态的一种体现。

【例 8.14】不使用多态的面向对象程序实现

```
#例 8.14 不使用多态的面向对象程序实现
class ArmyDog (object):
    def bite_enemy (self):
        print("追击敌人。")
class DrugDog(object):
    def track_drug(self):
        print("追查毒品。")
class Person (object):
    def work_with_army(self, dog):
        dog.bite_enemy()
    def work_with_drug (self, dog):
        dog.track_drug()
person=Person ()
person.work_with_army(ArmyDog())
person.work_with_drug(DrugDog())
```

运行结果：

```
追击敌人。
追查毒品。
```

接下来是使用多态的示例。

【例 8.15】使用多态的面向对象程序实现

```
#例 8.15 使用多态的面向对象程序实现
class Dog (object):
    def work(self):
        pass
class ArmyDog(Dog):
    def work(self):
        print("追击敌人。")
class DrugDog(Dog):
    def work(self):
        print("追查毒品。")
class Person (object):
    def work_with_dog(self, dog):
        dog.work()
person= Person()
person.work_with_dog(ArmyDog())
person.work_with_dog(DrugDog())
```

运行结果：

```
追击敌人。
追查毒品。
```

8.6 封装

封装数据的主要原因是保护隐私。在程序设计中，封装(Encapsulation)是对具体对象的一种抽象，即将某些部分隐藏起来，在程序外部看不到，其含义是其他程序无法调用。要了解封装，离不开“私有化”，就是将类或者函数中的某些属性限制在某个区域之内，外部无法调用。Python 通过在变量名前加双下画线来实现“私有化”。如__privatedata=0，定义私有方法则是在方法名称前加上下画线_。


扫码看视频 8.5

封装其实分为两个层面，第一个层面的封装：创建类和对象会分别创建二者的名称空间，只能用“类名.”或者“实例名.”的方式去访问里面的属性和方法，这就是一种封装。

第二个层面的封装：类中把某些属性和方法隐藏起来(或者说定义成私有的)，只在类的内部使用，外部无法访问，或者留下少量接口(函数)供外部访问。

在继承中，父类如果不想让子类覆盖自己的方法，可以将方法定义为私有的。首先，正常访问情况如下。

【例 8.16】封装性示例

```
#例 8.16 封装性示例
class A:
    def fm(self):
        print("from A")
    def test(self):
        self.fm()
class B(A):
    def fm(self):
        print("from B")
b=B()
b.test()
```

输出结果如下

```
from B
```

接下来，将 fm()定义成私有的，即__fm()，输出结果变成了“from A”，这意味着子类 B 没有覆盖掉父类 A 的__fm()方法。

【例 8.17】私有方法并不会被覆盖的封装示例

```
#例 8.17 私有方法并不会被覆盖的封装示例
class A:
    def __fm(self):
        print("from A")
    def test(self):
        self.__fm()
class B(A):
    def __fm(self):
        print("from B")
b = B()
b.test()
```

运行结果：

```
from A
```

8.7 单例模式


扫码看视频 8.6

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。

在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

【例 8.18】创建一个全局唯一的对象实例

```
#例 8.18 创建一个全局唯一的对象实例
#例 8.18 创建一个全局唯一的对象实例
class Earth(object):
    __instance = None   # 定义一个类属性做判断
    def __new__(cls):
        if cls.__instance == None:
            # 如果__instance 为空证明是第一次创建实例
            # 通过父类的__new__(cls)创建实例
            cls.__instance = object.__new__(cls)
            return cls.__instance
        else:
            # 返回上一个对象的引用
            return cls.__instance
a = Earth()
print(id(a))
b = Earth()
print(id(b))
```

运行结果：

```
34961280
34961280
```

在 Python 中，一个类创建对象实例是通过调用父类 object 的__new__(cls)方法来创建对象的。我们可以通过重写__new__(cls)方法去实现类只创建一个实例。

8.8 实验：人类与宠物模拟世界

【例 8.19】创建一个 Animal 类，封装了基本的属性和方法，使用继承创建 Dog 类，Cat 类，Person 类，并调用这些类。

```
class Animal:
    # 所用的知识 Animal 类的封装 -> Dog 类，Cat 类，Person 类的继承->多态
    # 把所有的共同属性和方法封装在一个公有类里面让子类继承父类的方法来实现和数据
    # 在创建一个小狗实例的时候，给它设置几个属性
    def __init__(self, name, age=1):
        self.name = name
        self.age = age
    def eat(self):
        # print("名字是%s，年龄%d 岁的小狗在吃饭"%(self.name,self.age))
        print("%s 吃饭"%self)
```

```
                return self
        def play(self):
                print("%s 玩"%self)
                return self
        def sleep(self):
                print("%s 睡觉"%self)
                return self
class Dog(Animal):
        def work(self):
                print("%s 看家"%self)
        def __str__(self):
                # self 对象本身对字符串的一个描述
                return "名字是{}，年龄{}岁的小狗在".format(self.name,self.age)
class Cat(Animal):
        def work(self):
                print("%s 捉老鼠"%self)
        def __str__(self):
                # self 对象本身对字符串的一个描述
                return "名字是{}，年龄{}岁的小猫在".format(self.name, self.age)
class Person(Animal):
        def __init__(self, name, pets, age=1):
                super(Person,self).__init__(name,age)
                self.pets = pets
        def feed_pets(self):
                # 所用的知识就是多态，养宠物，和让宠物工作也都是多态
                for pet in self.pets:
                        pet.eat()
                        pet.sleep()
                        pet.play()
        def make_pets_work(self):
                for pet in self.pets:
                        pet.work()
        def __str__(self):
                # self 对象本身对字符串的一个描述
                return "名字是{}，年龄{}岁的人在".format(self.name, self.age)
# d = Dog("小黑",18)
# c = Cat("小红",2)
# p = Person("BruceLong", [d, c], 24 )
# print(p.__dict__)
d = Dog("小黑",18)
#  selr 中谁调用就是谁 此处 d 会去 Animal 中找到 self 和里的的属性和方法而 Animal 里的 self 就是 Dog 类
c = Cat("小红",2)
p = Person("BruceLong", [d, c], 24 )
```

```
p.feed_pets()
p.make_pets_work()
```

运行结果：

```
名字是小黑，年龄 18 岁的小狗在吃饭
名字是小黑，年龄 18 岁的小狗在睡觉
名字是小黑，年龄 18 岁的小狗在玩
名字是小红，年龄 2 岁的小猫在吃饭
名字是小红，年龄 2 岁的小猫在睡觉
名字是小红，年龄 2 岁的小猫在玩
名字是小黑，年龄 18 岁的小狗在看家
名字是小红，年龄 2 岁的小猫在捉老鼠
```

第九单元：文件操作

文件是指存放在外部存储介质（可以是磁盘、光盘、磁带等）上一组相关信息的集合。它用于永久地将数据存储在非易失性的内存中(例如：硬盘、U 盘、移动硬盘、光盘等)将数据长期保存成文件，在需要的时候使用。

在 Windows 系统中，文件可以是文本文档、图片、程序等，每种类型的文件其扩展名也不同(例如：txt、jpg、py 等)。而在 Linux 操作系统中，一切皆是文件。对于 Python 而言，文件是一种类型对象，像前面介绍的其他类型(例如：str)一样，可以按照不同的标准进行如下分类。

（1）根据文件依附的介质不同，可以将文件分为普通文件和设备文件。

普通文件：指驻留在磁盘或其他外部介质上的一个有序数据集。

设备文件：指与主机相连的各种外部设备，将外部设备当作文件来处理。

（2）根据文件的组织形式，可以将文件分为顺序存取文件和随

机存取文件。

顺序读写文件(Sequential Access File) ：是指按从头到尾的顺序读出或写入的文件。

随机读写文件(Random Access file) ：每个记录的长度是相同的，因而通过计算便可直接访问文件中的特定记录，是一种跳跃式直接访问方式。

（3）按文件存储数据的形式，又可以将文件分为 ASCII 文件和二进制文件。

ASCII 文件：又称文本文件，ASCII 码文件中每个字节存放一个 ASCII 代码，代表一个字符，此种存储形式便于输出显示，在 DOS 操作系统下可以直接阅读。

二进制文件：二进制文件中的数据是按照在内存中的二进制存储格式存放的，此种存储形式节省存储单元。二进制文件在 DOS 操作系统下不能直接阅读。

例如：将整数 1949 分别存储在这两种数据文件中，其表现形式有所不同。以 ASCII 码文件存储，占用 4 个字节。

00110001	00111001	00110100	00111001
'1'	'9'	'4'	'9'

以二进制补码文件存储，占用 2 个字节。

00000111	10011101

9.1 Python 中文件的打开和关闭

计算机对文件进行处理时，首先把文件读入内

扫码看视频 9.1

存，然后在内存中对文件进行处理，再将处理的结果写入文件，最后关闭文件。

(1)打开文件(open)：建立用户程序与文件的联系，为文件分配一个文件缓冲区，声明文件的地址、名称及文件处理方式等。

(2)文件读/写操作(read/write)：对文件的读、写、追加和定位等操作。从磁盘将数据缓冲到内存的过程称为“读”文件，从内存将数据存到磁盘的过程称为“写”文件。

(3)关闭文件(close)：切断文件与用户程序的联系，把文件缓冲区的数据全部写入磁盘，释放掉该文件占用的缓存区空间，以免造成数据丢失。

9.1.1 打开文件

使用 Python 内置 open()函数可以打开指定的文件， open 函数的语法格式如下：

open(filename [,mode] [,encoding])

open()函数最常用的函数有 3 个，分别是 filename(文件名称)、mode(文件打开模式)和 encode(文件编码方式)。其中，filename 不可以省略，其他参数都可以省略，省略时会使用默认值。另外还有 5 个不常用函数。

1．默认方式打开文件

open()函数默认以只读方式打开文件，并且返回文件对象。

假如 studentinfo. txt 文本文件中有以下内容：

```
张三
李四
```

```
我们都喜欢 Python。
```

【例 9.1】文件基本操作(open)。准备一个名为“myfile. txt”的文本文件，使用 Python 内置 open()函数打开文件，代码如下所示：

```
#例 9.1 文件基本操作(open())
#打开当前目录中的文件
f1= open("studentinfo.txt")
f1.close()
#指定完整路径
f2= open("C:\ch9\studentinfo.txt")
f2.close()
```

2．使用指定方式打开文件

打开文件时如果不指定模式，那么默认为 r，以只读方式打开文件。此外，还可以显式指定打开模式：读取使用 r、写入使用 w、追加使用 a。还可以指定以文本模式或二进制模式打开文件，处理非文本文件，例如图像、EXE 文件等时通常使用二进制模式。

【例 9.2】文件基本操作(打开模式)

```
#例 9.2 文件基本操作(打开模式)
#相当于'r'或'rt
f1= open("studentinfo.txt")
print("文件打开模式为：  ",f1.mode)
f1.close()
#以文本模式写入
f2=open("studentinfo.txt",'w')
f2.close()
#以二进制模式读写
f3= open ("studentinfo.jpg",'r+b')
f3.close()
```

运行结果：

```
文件打开模式为：  r
```

在打开文件时指定的打开模式的详细解释如表 9-1 所示。

访问模式	说明
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模

	式。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
b	二进制模式
t	文本模式，也是系统默认的文件打开方式。
+	打开一个用于更新（读取和写入）的文件。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件

	用于读写。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

下面的例子打开 studentinfo. txt 文件，并逐一打印出文件中的各行内容，然后调用 close()。通过文件对象遍历时，每次读取一行。

【例 9.3】用只读方式打开文件并遍历输出结果

```
#以只读模式打开 studentinfo. txt 文件，并读取其内容。
f1=open("studentinfo.txt",'r')
for line in f1:
    print(line)
f1.close()
```

运行结果：

```
张三
李四
我们都喜欢 Python。
```

3.指定编码类型

文件默认的编码依赖于平台。在 Windows 系统中，默认编码为 CP936；在 Linux 系统中，默认编码为 UTF-8。代码在不同平台上将会有不同表现。因此，当以文本模式处理文件时，要指定编码类型。

【例 9.4】以特定编码方式打开文件

```
#例 9.4 以特定编码方式打开文件
#文件默认编码
f1= open ("studentinfo.txt")
print("文件编码方式为：",f1.encoding)
f1.close()
#打开文件时指定编码类型
```

```
f2= open("studentinfo.txt",mode='r',encoding='utf-8')
f2.close()
```

运行结果：

```
文件编码方式为：  cp936
```

文件对象常见的属性如下：

文件对象属性	含义
name	返回文件的名称
mode	返回文件的打开方式
closed	如果文件被关闭返回 True，否则返回 False

9.1.2 关闭文件

程序完成对文件的操作后，需要使用 close()来主动关闭文件，以释放与该文件绑定的资源。我们可以采取常规关闭方式、异常处理关闭方式和使用 with 语句关闭方式三种方法来完成关闭操作。

(1)常规方式：使用 close()方法来完成关闭文件。这种方式并不完全安全，很有可能会引发 IOError。其语法格式如下：

文件对象名.close()

(2)异常处理方式：使用 try…finally 块来关闭文件。即使出现异常，也可以确保文件能够被正确地关闭。

(3)使用 with 语句方式：使用 with 语句可以确保文件被安全地关闭，该动作是在内部完成关闭文件的操作，其效果与 try…finally 块是一样的，而且无须显式地调用 close()，代码更简洁优雅。

三种关闭文件方式如例 9-5 所示。

【例 9.5】三种方式关闭文件

```
#例 9.5 三种方式关闭文件
```

```
#(1)常规方式
print("(1)常规方式:")
f1= open("studentinfo.txt")
f1.close()
#(2)异常处理方式
print("(2)异常处理方式:")
try:
    f2=open("studentinfo.txt")
finally:
    f2.close()
#(3)  使用 with 语句方式
print("(3)使用 with 语句方式:")
with open("studentinfo.txt") as f3:
    pass
```

运行结果：

```
(1)常规方式:
(2)异常处理方式:
(3)使用 with 语句方式:
```

9.2 文件的读/写操作


扫码看视频 9.2

文件读就是从文件中读出数据到内存中去；文件写就是把内存中的数据写入文件中。读和写使用的读写语句不一定相同。

9.2.1 文件的读操作

Python 中，读取文本文件的内容必须以 r 模式打开文件。读取文本文件有 read()、readline()、readlines()三种常用方法。

1.read()方法

read()方法一次性读取文件的所有内容，并存放在一个大字符串中。其语法格式如下：

```
文件对象.read([size])
```

【例 9.6】使用 read()方法读取文本文件

```
#例 9.6 使用 read()方法读取文本文件
with open ("studentinfo.txt", mode='r', encoding='utf-8') as f1:
```

```
print(f1.read (2))
print(f1.read (2))
print(f1.read (6))
print(f1.read (8))
```

运行结果：

```
张三
李四
我们都喜欢
Python。
```

2.readline()方法

readline()方法逐行读取文本，结果是一个 list。其语法格式如下：

```
文件对象.readline()
```

【例 9.7】使用 readline ()方法读取文本文件

```
#例 9.7 使用 readline ()方法读取文本文件
with open("studentinfo.txt" mode='r') as f1:
    line=f1.readline()
    while line:
        line=line.rstrip('\n')
        print (line)
        line=f1.readline()
```

运行结果

```
张三
李四
我们都喜欢  Python。
```

3.readlines()方法

```
文件对象.readlines()
```

readlines()方法一次性读取文本的所有内容，结果是一个 list。readlines()读取的文件内容中，每行末尾都会带一个“\n”换行符。

【例 9.8】使用 readlines ()方法读取文本文件

```
#例 9.8 使用 readlines()方法读取文本文件
with open("studentinfo.txt",mode='r') as f1:
    for line in f1.readline():
        line_str=line.rstrip('\n')
        print(line_str)
```

运行结果

```
张三
李四
我们都喜欢 Python。
```

9.2.2 文件的写操作

Python 中，为文本文件写入内容必须以 w 模式打开文件,追加文件内容则使用 a 模式打开。为文件写入内容有 write()和 writelines()两种常用方法。

（1）write()方法

write()方法将参数内容写到文件中，write()方法不会追加一个“\n”换行符。它返回写入的字符个数。其语法格式如下：

```
文件对象. write (字符串)
```

【例 9.9】使用 write()方法向文本文件写入内容

```
#例 9.9 使用 write()方法向文本文件写入内容
#写操作
with open("teacher.txt",mode='w') as f1:
    f1.write("我会使用 Python 程序设计。")
    f1.write("\n")
    f1.write("让我来教大家学习 Python 程序设计。")
    f1.write("\n")
    #读操作
with open("teacher.txt",mode='r') as f2:
    for line in f2.readlines ():
        line_str=line.rstrip('\n')
        print(line_str)
```

运行结果：

```
我会使用 Python 程序设计。
让我来教大家学习 Python 程序设计。
```

2. writelines()

writelines()把多行内容写到文件中，参数可以是一个可迭代的对象、列表、元组等。其语法格式如下：

```
文件对象. writelines (字符串元素的列表)
```

【例 9.10】使用 writelines ()方法向文本文件写入内容

```
#例 9.10 使用 writelines ()方法向文本文件写入内容
#写操作
list=("先易后难。","\n","循序渐进。","\n")
with open("teacher.txt",mode='a') as f3:
    f3. writelines(list)
#读操作
with open("teacher.txt",mode='r') as f4:
    for line in f4. readlines():
        line_str=line.rstrip('\n')
        print(line_str)
```

运行结果：

```
我会使用 Python 程序设计。
让我来教大家学习 Python 程序设计。
先易后难。
循序渐进。
```

9.3 文件和目录操作

Python 有一个 os 模块，提供了许多便利的方法来管理文件和目录。os 提供了创建目录、删除目录、删除文件、执行操作系统命令等方法，使用时必须导入 os 包。


扫码看视频 9.3

9.3.1 os.remove()方法

remove()方法用于删除指定文件，一般都会结合 os.path.exists()方法使用，即先检查该文件是否存在，再删除该文件。

【例 9.11】使用 remove()方法来删除文件

```
#例 9.11 使用 remove()方法来删除文件
import os
file_name="test.txt"
if os.path.exists(file_name):
    os.remove(file_name)
    print(file_name+"文件删除成功！")
else:
    print(file_name+"文件未找到！")
```

运行结果：

```
test.txt 文件删除成功!
```

第二次运行本程序结果：

```
test.txt 文件未找到!
```

9.3.2 os.mkdir()方法

用 mkdir()方法可以创建指定名称的目录。执行后会在当前目录创建对应的目录。但如果目录已经创建，执行时就会产生错误。所以一般要先用 os.path.exists()方法检查该目录是否存在，再决定是否要创建该目录。一般都会结合 os.getcwd()方法使用，即先查看当前目录位置，检查该所要创建的目录是否存在，再创建该目录。

【例 9.12】使用 mkdir()方法来创建目录

```
#例 9.12 使用 mkdir()方法来创建目录
import os
dir_str=os.getcwd()
my_dir="PythonFile"
if not os.path.exists(my_dir):
    os.mkdir(my_dir)
    print("当前目录为："+ dir_str +"。在该目录下，"+my_dir +"目录创建成功！")
else:
    print("当前目录为："+ dir_str +"。在该目录下，"+my_dir +"目录已存在！")
```

运行结果：

```
当前目录为：D:\ws\python。在该目录下，PythonFile 目录创建成功!
```

9.3.3 os.rmdir()方法

rmdir()方法可以删除指定目录，删除目录前必须先删除该目录中的文件。一般都会先检查目录是否存在，再删除该目录。

【例 9.13】使用 rmdir()方法来删除目录

```
#例 9.13 使用 rmdir()方法来删除目录
import os
dir_str=os.getcwd()
my_dir="PythonFile"
if os.path.exists(my_dir):
    os.rmdir(my_dir)
```

```
        print(my_dir +"目录删除成功！")
    else:
        print(my_dir +"目录未找到！")
```

运行结果：

```
PythonFile 目录删除成功!
```

9.3.4 os.system()方法

system()方法用来执行操作系统命令，例如：清除屏幕、创建目录，复制文件等等。

【例 9.14】使用 system ()方法来完成文件或目录操作

```
#例 9.14 使用 system ()方法来完成文件或目录操作
import os
dir_str=os.path.dirname(__file__)
os.system("cls")
os.system("mkdir PythonFile2")
os.system("copy teacher.txt PythonFile2\teacher2.txt")
file_name= dir_str+"PythonFile2\teacher2.txt"
os.system("notepad "+file_name)
```

运行结果：可看到打开了新复制的 teacher2.txt 文件。

9.3.5 os.rename ()方法

rename()方法用来为文件重命名。它需要两个参数，即当前的文件名和新文件名。

【例 9.15】使用 rename ()方法来为文件重命名

```
#例 9.15 使用 rename ()方法来为文件重命名
import os
file_name="test.txt"
file_name_new="test_new.txt"
if os.path.exists(file_name):
    os.rename (file_name,file_name_new)
    print("文件重命名为：  "+file_name_new)
else:
    print(file_name+"文件未找到！")
```

运行结果：

```
文件重命名为：  test_new.txt
```

9.3.6 os.walk()方法

os.walk()方法用来搜索指定目录及其子目录，它返回一个包含 3 个元素的元组（dirpath,dirname,filenames）：

dirpath：以字符串形式返回该目录下所有的绝对路径；

dirname：以列表形式返回每一个绝对路径下的目录；

filenames：以列表形式返回该路径下的所有文件。

【例 9.16】使用 walk()方法来搜索指定目录及其子目录

```
#例 9.16 使用 walk()方法来搜索指定目录及其子目录
import os
cur_path=os.path.dirname(__file__)
cur_path+="/PythonFile"
sample_tree=os.walk(cur_path)
for dir_name, sub_dir, files in sample_tree:
    print("文件路径：",dir_name)
    print("目录列表：",sub_dir)
    print("文件列表：",files)
```

运行结果：

```
文件路径：  C:/Users/Administrator/PycharmProjects/ch9/PythonFile
目录列表：  []
文件列表：  ['teacher.txt']
```

9.4 CSV 文件操作


扫码看视频 9.4

CSV 文件不是一种独立的文件类型，而是一种有特殊格式的文本文件。CSV 文件用于纯文本存储表格数据。由于数字表格大量使用 Excel 进行处理，所以也经常用 Excel 处理 CSV 文件。这种文件格式在绝大多数计算机平台上都通用，所以很有必要掌握 CSV 文件的处理。

采用逗号分割的存储格式叫作 CSV(Comma-Separated Values，逗号分隔值)格式，它是种通用的、相对简单的文件格式，在商业和科

学上广泛应用，大部分编辑器都支持直接读入或保存文件为 CSV 格式。根据数据的关系不同，数据组织可以分为一维数据、二维数据和高维数据。

9.4.1 CSV 文件的一维数据读写

一维数据保存成 CSV 格式后，各元素采用逗号分隔，形成一行。从 Python 表示到数据存储，需要将列表对象输出为 CSV 格式以及将 CSV 格式读入成列表对象。

列表对象输出为 CSv 格式文件方法如例 9.15 所示，采用字符串的 join 方法最为方便。

【例 9.17】将一维数据写入 CSV 格式文件

```
#例 9.17 将一维数据写入 CSV 格式文件
list=['苹果','葡萄','香蕉','西瓜']
f1=open("shuiguo.csv", "w")
f1.write(",".join(list)+ "\n")
f1.close()
```

运行结果：在当前目录下会创建 shuiguo.csv 文件，并将列表内容写入到该 csv 文件中。

从 CSV 文件中读取数据首先需要打开 CSV 格式文件，读取一维数据并将其表示为列表对象。

【例 9.18】从 CSV 格式文件读取一维数据

```
#例 9.18 从 CSV 格式文件读取一维数据
f2=open("shuiguo.csv","r")
list=f2.read().strip("\n").split(",  ")
f2.close()
print(list)
```

运行结果

```
['苹果,葡萄,香蕉,西瓜']
```

9.4.2 CSV 文件的二维数据读写

二维数据是由多条一维数据构成的，可以看成一维数据的组合形式。因此，二维数据可以用二维列表来表示，即列表的每个元素对应二维数据的一行，这个元素本身也是列表类型，其部各元素对应这行中的各列值。

【例 9.19】将二维数据写入到 CSV 格式文件

```
#例 9.19 将二维数据写入到 CSV 格式文件
list =[['水果名称', '单价', '数量', '金额'],
['葡萄', '30', '1.5', '45'],
['香蕉', '10', '3', '30'],
['西瓜', '12', '2', '26']]
f3_name="shuiguoqingdan.csv"
f3=open(f3_name, 'w')
for row in list:
    f3.write(",".join(row)+ "\n")
print("数据写入",f3_name, "成功。")
f3.close()
```

运行结果

```
数据写入 shuiguoqingdan.csv 成功。
```

借鉴例 9.18 的方法，从 CSV 文件中读取数据首先需要打开 CSV 格式文件，读取二维数据并将其表示为列表对象。

【例 9.20】从 CSV 格式文件读取二维数据

```
#例 9.20 从 CSV 格式文件读取二维数据
f4_name="shuiguoqingdan.csv"
f4=open(f4_name, "r")
list=[]
for line in f4:
    list.append(line.strip('\n').split(","))
f4.close()
print(list)
```

运行结果

```
[['水果名称', '单价', '数量', '金额'], ['葡萄', '30', '1.5', '45'], ['香蕉', '10', '3', '30'], ['西瓜', '12', '2', '26']]
```

第十单元：Python 异常处理

Python 中（至少）有两种错误：语法错误和异常（ syntax errors 和 exceptions ）。语法错误通常是由于我们没有正确掌握语法或输入代码过程中出错而造成的。我们可以在编码和输入过程中尽力避免。异常是在程序执行过程中发生的一个事件，会影响了程序的正常执行。

10.1 异常的概念

10.1.1 语法错误


扫码看视频 10.1

语法错误，也称作解析错误，也许是学习 Python 过程中最常见的。下面的代码中 print 有拼写错误，程序运行会报错。

```
prin("hello,world")
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/ch10/ex10.1.py", line 1, in <module>
    prin("hello,world")
NameError: name 'prin' is not defined
```

语法分析器指出错误行号为第 1 行，因为 print 少了一个“t”。错误会输出文件名和行号，方便查找发生错误的位置。

这类错误需要编程者自己不断提高编辑和编程水平，以减少发生的频率，Python 系统无法帮我们解决这类的问题。

10.1.2 异常

即使语句或表达式在语法上是正确的，当试图执行它时也可能会引发错误。当 Python 检测到一个错误时，解释器就会指出当前流已无法继续执行下去，运行期检测到的错误即为异常。异常是指因为程

序出错而在正常控制流以外采取的行为。

下面的代码用于计算用户的体重身高指数（BMI），程序会根据用户输入的身高和体重数据计算出用户的 BMI 并显示。

【例 10.1】体重身高指数（BMI）程序异常

```
#例 10.1 体重身高指数（BMI）程序异常
height=float(input("请输入您的身高(单位：m): "))
weight=float(input("请输入您的体重(单位：Kg):"))
bmi=round((weight/(height*height)),2)
print("您的 BMI 指数为: ",bmi)
```

为了得到人们想要的程序结果，要求我们输入合理的数据必须合理，一旦用户误将身高数据输入为 0，则程序会出现异常而退出，并可以看到错误信息如下所示：

```
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/ch10/ex10.1.py", line 4, in <module>
    bmi=round((weight/(height*height)),2)
ZeroDivisionError: float division by zero
```

在 Python 程序设计中，上例中的异常被称为零除错误（ZeroDivisionError）。

10.2 异常类

Python 的异常处理能力是很强大的，它有很多内置异常，可向用户准确反馈出错信息。在 Python 中，异常也是对象，可对它进行操作。BaseException 是所有内置异常的基类，但用户定义的类并不直接继承 BaseException，所有的异常类都是从 Exception 继承，且都在 exceptions 模块中定义。Python 自动将所有异常名称放在内建命名空间中，所以程序不必导入 exceptions 模块即可使用异常。一旦引发而且没有捕捉 SystemExit 异常，程序执行就会终止。如果交互式会话遇到一个未被捕捉的 SystemExit 异常，会话就会终止。

常见异常类如下：

1.TypeError（类型错误）：必须是一个字符串　不能是数字

错误演示：

```
name = '熊晨'
age = 20
print('我的名字叫'+name+',我的年龄是'+age)
```

解决方案：使用‘+’拼接的时候，必须使用字符串，或者将数字转化为字符串

2.IndentationError（缩进错误）：未知缩进不匹配任何缩进等级

错误演示：

```
for index in range(10):
    if name == '小王':
        print('hello')
else:
        print('nothing')
```

解决方案：tab 自动缩进

3.IndexError（索引错误）：字符串超出了范围

错误演示：

```
str = 'good idea'
print(str[20])
```

解决方案：查看字符串的长度，索引要小于长度

4.SyntaxError（语法错误）：非法的语法

错误演示：

```
name = '小陈'
if name = '小陈':
    print('hello')
```

解决方案：赋值符合不能用于条件判断

5.ValueError（值错误）：子字符串未找到

错误演示：

```
str = 'hello world'
result = str.index('c')
print(result)
```

解决方案：输入字符串内的字符

6.AttributeError（属性错误）：元组对象没有属性‘remove’

错误演示：

```
tp1 = ((),[],{},1,2,3,'a','b','c',3.14 ,True)
tp1.remove(1)
print(tp1)
```

解决方案：元组对象没有 remove()方法，将元组改为列表，就可以解决这个数据类型不匹配的问题。

7.KeyError（key 键错误）：没有指定的键

错误演示：

```
dic1 = {
    'name': '小许',
    'age' : 17 ,
    'friend':['嘻嘻','嚷嚷','慌慌','张张','欣欣','向荣']
}
print(dic1['fond'])
```

解决方案：给字典中指定的键赋值，如果有这个键则重新修改这个键对应的值，如果没有这个键则创建这个键并且设置对应的值。

10.3 异常处理

扫码看视频 10.2

程序设计中经常需要考虑对程序各种异常情况进行预判和处理。程序中的异常情况有很多种，前面已经做了一些介绍。合理的处理方式是当异常发生时程序要处理它，并提示用户输入正确格式的数字。

在 Python 程序设计中，异常处理语法结构如下：

```
try:
    <代码块>
except<Exception Type1>:
    <异常处理代码块 1>
except:
    <异常处理代码块 2>
else:
    <异常处理代码块 3>
finally:
    <异常处理代码块 4>
```

try 关键字告诉系统要开始监测异常了，<代码块>是可能出现异常的代码块，可能有一行或多行。当一个异常出现时，<代码块>中剩余的代码将被跳过，不再执行。

except 语句分支可以有一条或多条， ExceptionType 是具体的异常类型，可以是系统内置异常或用户自定义异常。

当异常发生时，程序根据发生的异常类型在 except 语句分支中依次进行匹配，如果匹配成功，刚执行此 except 后面的异常处理语句块，执行完成后退出整个异常处理语句。

如果匹配不成功，则转到不标明异常类型的最后一个 except 语句分支。

如果<代码块>执行时未发生异常，则执行 else 后面的语句块。

不管程序是否发生异常，都会执行 finally 后面的语句分支。

else 子句、 finally 子句和不标明异常类型的最后一个 except 语句分支都是可选项，且最多只有一个。

【例 10.2】处理打开文件异常

```
#例 10.2 处理打开文件异常
try:
    f1= open("test.txt" "r")
    readstr=f1. read(20)
```

```
except IOError:
    print("没有找到文件或读取文件失败")
else:
    print(readstr)
    f1. close()
```

执行上面这段程序时，如果所读取 test.txt 文件不存在，程序产生异常，此异常被 except 语句分支捕获，程序将输出：没有找到文件或读取文件失败。如果文件存在，则程序会顺利打开文件并执行 else 语句分支，输出文件中的前 20 个字符，然后关闭文件，最后程序退出。

有时不清楚可能产生的异常类型，程序员也可以不指定异常类型，即将 except 分支中的错误类型去掉。

【例 10.3】使用不指定异常类型的 except 语句处理异常

```
#例 10.3 使用不指定异常类型的 except 语句处理异常
try:
    num=int( input("请输入一个数值："))
    print("您输入的数值是：",num)
except:
    print("您输入的不是合法的数据，请重新输入。")
```

运行结果：

```
请输入一个数值：a
您输入的不是合法的数据，请重新输入。
```

第二次再运行程序，输入一个数字，结果为：

```
请输入一个数值:8
您输入的数值是 8
```

尽管这种偷懒的方式在本例能正常工作，但不推荐这样做。在编程过程中我们要尽可能清楚哪里可能发生异常、发生什么类型的异常，对具体的异常进行具体的处理。

【例 10.4】使用 else 子句异常处理

```
#例 10.4 使用 else 子句异常处理
```

```
import sys
try:
    f=open('test.txt')
    s= f.readline()
    i=int(s.strip())
except OSError as err:
    print ("操作系统错误：{}".format(err))
except ValueError:
    print ("不能将数据转换成整数。")
else:
    print("无法预知的错误："+sys.exc_info()[0])
```

运行结果：

```
操作系统错误：[Errno 2] No such file or directory: 'test.txt')
```

异常处理时，finally 字句也是可选的且有用的。

【例 10.5】使用 finally 结束程序执行

```
#例 10.5 使用 finally 结束程序执行
try:
    height=float(input("请输入您的身高(单位：m): "))
    weight=float(input("请输入您的体重(单位：Kg):"))
    bmi=round((weight/(height*height)),2)
except ValueError:
    print("你输入的数字有误。\n")
except ZeroDivisionError:
    print("0 不能作为除数使用，请输入正确的数字。\n")
else:
    print("您的 BMI 指数为: ",bmi)
finally:
    print("程序退出。")
```

运行程序，输入字母 a，结果为：

```
你输入的数字有误。
print(“程序退出。”)
```

第二次运行程序，输入体重 60，输入身高 0，结果为：

```
0 不能作为除数使用，请输入正确的数字。
print(“程序退出。”)
```

第三次运行程序，输入体重 60，输入身高 1.75，结果为：

```
您的 BMI 指数为:19.59
print(“程序退出。”)
```

此段代码在 try 分支中包含两种可能产生的异常，分别是用户输

入了不正确的数据类型、身高输入为 0。身高和体重显然要求输入数值类型，如果输入了不正确的数据类型，则产生 ValueError 异常，如果身高输入为 0，则在执行除法运算时产生 ZeroDivisionError 异常。这两个异常分别被两个 except 分支捕获并处理。如果没有产生异常，则执行 else 分支，输出用户的 BMI 指数。不管是否产生异常，都会执行 finally 分支，输出“ 程序退出”。

虽然一个 try 语句可以有多个 except 子句，用来明确地处理不同的异常，但每次运行至多只有一个异常处理子句会被执行。如果无匹配的异常类型，except 子句都不执行。异常处理子句只处理对应的 try<代码块>中发生的异常。

10.4 抛出异常


扫码看视频 10.3

Python 可以自动引发异常，也可以通过 raise 显式地抛出异常。一旦执行了 raise 语句，raise 后面的语句将不能执行。换句话说， ralse 语句允许程序员在任何必要的时候强制抛出一个指定的异常，而不必等 Python 引发。其语法格式如下：

```
raise exceptionName
```

只要在 raise 关键字后跟上一个异常类型名，就可立即引发一个异常，改变程序的执行路径。

【例 10.6】使用 raise 主动抛出异常

```
#例 10.6 使用 raise 主动抛出异常
data1=input("请输入一个整数：")
try:
    if data1.isdigit():
        data1_int=int(data1)
```

```
        else:
            raise ValueError
    except ValueError:
        print("将数据转换成整数时出错：", data1)
```

运行程序，输入字母 a，结果为：

```
将数据转换成整数时出错：  a
```

以上代码在 try 语句块中并没有自发产生异常的代码，而是自行根据条件判断情况抛出了一个异常，后面的 except 分支会处理抛出的异常。

【例 10.7】使用 raise 改变程序执行的流程

```
#例 10.7 使用 raise 改变程序执行的流程
try:
    for i in range(3):
        for j in range(3):
            if i== 2:
                raise
            print(i,j)
except:
    print("当 i 的值为 2 时，程序循环结束。")
```

运行程序，输入字母 a，结果为：

```
00
01
02
10
11
12
当 i 的值为 2 时，程序循环结束。
```

以上例子中的 raise 语句后面并没有跟一个异常名，用于控制程序执行流程。

10.5 断言

扫码看视频 10.4

断言（assert）语句用来声明某个条件是真的，其作用是测试一个条件(condition)是否成立，如果

不成立，则抛出异常。断言的语法格式如下：

```
assert condition[,expression]
```

如果 condition 为 false，就 raise 一个描述为 expression 的 AssertionError 出来。expression 可以省略。

【例 10.8】使用 assert 断言示例

```
#例 10.8 使用 assert 断言示例
v1 = 1
v2 = 2
assert (v1 < v2)
assert (v1 > v2),'{0} is not bigger than {1}'.format(v1,v2)
```

以上代码抛出异常，运行结果：

```
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/ch10/ex10.8.py", line 5, in <module>
    assert (v1 > v2),'{0} is not bigger than {1}'.format(v1,v2)
AssertionError: 1 is not bigger than 2
```

断言是用来检查非法情况而不是程序发生某种错误情况的工具，用来帮开发者快速定位问题的位置。异常处理用于对程序发生异常情况的处理，增强程序的健壮性和容错性。

对一个函数而言，一般情况下，断言用于检查函数输入的合法性，要求输入满足一定的条件才能继续执行，在函数执行过程中出现的异常情况使用异常来捕获。

10.6 用户自定义异常

扫码看视频 10.5

在 Python 程序中，用户可以创建新的异常类型。创建自定义异常就是创建一个异常子类。通过继承，用户可以命名它们自己的异常。自定义异常是通过扩展 BaseException 类或其子类来完成的。 BaseException 类是所有内置异常的基类，但用户定义的类并

不直接继承 BaseException 类，所有的异常类都是从 Exception 类继承，且都在 exceptions 模块中定义。Exception 类是常规异常的基类，也是 BaseException 类的子类之一。定义新异常类的语法格式如下：

```
class MyException(Exception)
pass
```

MyException 是用户自定义的异常类名，遵循变量命名规则。

【例 10.9】用户自定异常

```
#例 10.9 异常处理之自定义异常
class MyFirstError(Exception):
    def __init__(self,value):
        self.value= value
        def __str__(self):
            return repr(self.value)
try:
    raise MyFirstError(2*2)
except MyFirstError as mfe:
    print("发生了用户自定义异常，异常值为：",mfe.value)
```

运行结果：

```
发生了用户自定义异常，异常值为：  4
```

自定义异常类中，通常会定义几个新的属性信息，以供异常处理语句进行提取。如果一个新创建的模块中需要抛出几种不同的错误时，一个通常的做法是为该模块定义一个异常基类，然后针对不同的错误类型派生出对应的异常子类。

【例 10.10】用户自定异常基类，并继承此基类

```
#例 10.10 用户自定异常基类，并继承此基类
#自定义异常 需要继承 Exception
class MyException(Exception):
    def __init__(self, *args):
        self.args = args
#定义异常基类,然后在派生不同类型的异常
class loginError(MyException):
    def __init__(self, code=100,message= '登录异常',args = ('登录异常')):
        self.args = args
```

```
            self.message = message
            self.code = code
class loginoutError(MyException):
    def __init__(self):
            self.args = ('退出异常',)
            self.message = '退出异常'
            self.code = 200
try:
    raise loginError()
except loginError as loginError:
    print(loginError.code)          #输出错误代码
    print(loginError.message)    #输出错误信息
```

运行结果：

```
100
登录异常
```

与标准异常相似，大多数异常的命名都以“Eror 结尾。很多标准模块中都定义了自己的异常，用以报告在他们所定义的函数中可能发生的错误。

10.7 上下文管理


扫码看视频 10.6

with 是一种上下文管理协议，目的在于从流程图中把 try,except 和 finally 关键字和资源分配释放相关代码统统去掉，简化 try….except….finlally 的处理流程。with 通过 enter 方法初始化，然后在 exit 中做善后以及处理异常。所以使用 with 处理的对象必须有__enter__()和__exit__()这两个方法。其中__enter__()方法在语句体（with 语句包裹起来的代码块）执行之前进入运行，exit()方法在语句体执行完毕退出后运行。

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用

后自动关闭、线程中锁的自动获取和释放等。with 语法格式如下：

```
with expression [as target]:
    with_body
```

参数说明：

expression：是一个需要执行的表达式；

target：是一个变量或者元组，存储的是 expression 表达式执行返回的结果，可选参数。

如果某项工作完成后需要有释放资源或者其他清理工作，比如说文件操作时，就可以使用 with 优雅的处理，不用自己手动关闭文件句柄，而且 with 还能很好的管理上下文异常。

【例 10.11】打开一个文件并写入”Hello World”

```
#例 10.11 打开一个文件并写入"Hello World"
filename = 'my_file.txt'
mode = 'w'
writer = open(filename, mode)
writer.write('Hello ')
writer.write('World')
writer.close()
```

1-2 行，我们指明文件名以及打开方式(写入)。

第 3 行，打开文件，4-5 行写入“Hello world”，第 6 行关闭文件。

以上程序在正常情况下可以顺利执行。但是，如果程序出现异常，文件将无法正常执行完毕。当然，我们可以使用 try-finally 语句块来进行包装。

【例 10.12】打开一个文件并写入”Hello World”，需做异常处理

```
    #例 10.12 打开一个文件并写入"Hello World", 需做异常处理
    filename = 'my_file.txt'
    mode = 'w'
    writer = open(filename, mode)
    try:
```

```
        writer.write('Hello ')
        writer.write('World')
    finally:
        writer.close()
```

finally 语句块中的代码无论 try 语句块中发生了什么都会执行。因此可以保证文件一定会关闭。这样写当然没有，但如果程序代码更复杂，异常情况更多，try-finally 语句就会变得无能为力。例如我们要打开两个文件，一个读一个写，两个文件之间进行拷贝操作，那么通过 with 语句能够保证两者能够同时被关闭。

【例 10.13】打开一个文件并写入”Hello World”，使用 with 处理

```
#例 10.13 打开一个文件并写入"Hello World" ， 使用 with 处理
filename = 'my_file.txt'
mode = 'w'
with open(filename, mode) as writer:
    writer.write('Hello ')
    writer.write('World')
```

以上程序在正常情况下可以顺利执行。执行完毕后，当前目录下会生产一个 my_file.txt 文件，并在文件中写入 Hello World 的内容。

10.8 实验

【例 10.14】猜数字

```
#test  猜数字
#电脑随机生成 1~100 随机数, 用户输入一个数字, 电脑提示用户大或者小, 猜错, 继续提示；猜对，则程序终止。
#coding:utf-8
import random
num = random.randint(0,100)
#print(num)
while True:
        try:
                guess = int(input("请输入一个 1~100 的随机数:"))
        except ValueError as err:
                print("Enter 1~100", err)
                continue
        if guess > num:
```

```
                print("%d greater random number"    %(guess))
        elif guess < num:
                print("%d smaller random number" %(guess))
        else:
                print("guess bingo,game over!")
                break
        print("\n")
```

运行结果：

```
请输入一个 1~100 的随机数:22
22 smaller random number
```

【例 10.15】摸扑克牌

```
#摸扑克牌
from collections import namedtuple
Card = namedtuple('Card', ['ranks', 'suits'])
class FranchDeck:
    # 列表推导式生成纸牌所有的个数[2-A]
    ranks = [str(n) for n in range(2,11)] + list('JQKA')
    suits = ['红心', '方板', '梅花', '黑桃']   # 设置纸牌的花

    def __init__(self):
        self._cards = [Card(rank, suit) for rank in FranchDeck.ranks
                                                for suit in FranchDeck.suits]
    # self._cards = [所有的纸牌],得到一个所有纸牌的列表
    # for ranks in FranchDeck.ranks:
    #for suit in FranchDeck.suit:
    #Card(rank, suit)  等于  上面的写法

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, item):    # 抽取一张牌
        return self._cards[item]

deck = FranchDeck()
print(deck[0])
print(deck[51])
from random import choice
print(choice(deck))
print(choice(deck))
```

运行结果：

```
Card(ranks='2', suits='红心')
```

```
Card(ranks='A', suits='黑桃')
Card(ranks='5', suits='黑桃')
Card(ranks='Q', suits='红心')
```

第十一单元：Python 的模块使用与程序打包

模块是 Python 语言的一个重要概念，它可以将函数按功能划分到一起，以便日后使用或共享给他人。可以使用 Python 标准库中的模块，也可以下载和使用第三方模块。

11.1 模块的概述

在上几个单元中，我们知道面向对象开发时抽象为类的过程实际已经是一层封装了，而模块则是由变量、语句、函数或类的定义的程序文件组合而得，它的文件名字就是模块名加上.py 扩展名。一般来说用户编写程序的过程，也就是编写模块的过程。使用模块简单说来有那么 3 个优点。提高代码的可维护性；提高代码的可重用性；有利于避免函数名和变量名冲突。从抽象的视角来看，模块至少有三个角色：

- 代码的重用；
- 系统命名空间的划分；
- 服务和数据的共享。

模块函数和变量的使用方法：

可以使用下面的方式访问模块中的函数：

```
模块名.函数名(参数列表)
```

可以使用下面的方式访问模块中的变量：

```
模块名.变量
```

11.2 模块导入的 3 种方式

在 Python 中要使用模块必须将模块进行导入，基本每个 py 文件

中都会有 import 或者是 from import 语句来将模块进行导入。可是，这两种导入方法有什么不同，又该怎么用呢？

import 语句和 from … import 语句的基本语法

```
import module1[, module2[,... moduleN]
from modname import name1[, name2[, ... nameN]]
```

support.py 文件代码

```
# Filename: support.py

def print_func( par ):
    print ("Hello： ", par)
    return
```

test.py 引入 support 模块：

test.py 文件代码

```
# Filename: test.py

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Python 模块调用")
```

运行结果：

```
Hello ： Python 模块调用
```

一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。当我们使用 import 语句的时候，Python 解释器将通过搜索路径的方式，从一系列目录名中依次进行查找所引入的模块。

在上述 test.py 文件代码中，如果导入模块采用 from support import print_func，这种导入的方法不会把被导入的模块的名称放在当前的字符表中，而 Import 方法则是将整个模块对象赋值给一个变量名。From 将一个或多个变量名赋值给另外一个模块中同名的对象。

那么如果用方法二时，在 test.py 文件代码中再调用 support 中的 print_func（）函数，将不需要再把模块名称+函数名称来调用，可以直接调用函数，所以 from 容易污染命名空间。

第 3 种导入方式，"from … import * "语句把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

from modname import *

这提供了一个简单的方法来导入一个模块中的所有项目，在实践中, "from module import *" 不是良好的编程风格，如果使用 from 导入变量，且那些变量碰巧和作用域中现有变量同名，那么变量名就会被悄悄覆盖，因此不建议过多的使用第 3 种方式来声明。

11.3 Python 标准库中常用模块

11.3.1 sys 模块

sys 模块是 Python 标准库中最常用的模块之一。通过它可以获取命令行参数，从而实现从程序外部向程序传递参数的功能；也可以获取程序路径和当前系统平台等信息。


扫码看视频 11.1

【例 11.1】使用 sys.platform 获取操作系统平台信息

```
import sys
print(sys.platform)
#sys.platform 只返回操作系统的平台信息，并不包含操作系统的具体信息。
```

运行结果：

```
win32
```

【例 11.2】使用 sys.path 获取搜索模块的路径

```
import sys
print(sys.path);
```

运行结果：

```
['C:\\Users\\9day\\PycharmProjects\\untitled1',
'C:\\Users\\9day\\PycharmProjects\\untitled1',
'C:\\Users\\9day\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip',
'C:\\Users\\9day\\AppData\\Local\\Programs\\Python\\Python37\\DLLs',
'C:\\Users\\9day\\AppData\\Local\\Programs\\Python\\Python37\\lib',
'C:\\Users\\9day\\AppData\\Local\\Programs\\Python\\Python37',
'C:\\Users\\9day\\PycharmProjects\\untitled1\\venv',
'C:\\Users\\9day\\PycharmProjects\\untitled1\\venv\\lib\\site-packages',
'C:\\Users\\9day\\PycharmProjects\\untitled1\\venv\\lib\\site-packages\\setuptools-39.1.0-py3.7.egg',

'C:\\Users\\9day\\PycharmProjects\\untitled1\\venv\\lib\\site-packages\\pip-10.0.1-py3.7.egg']
```

从运行结果可以观察到，sys.path 是 python 的搜索模块的路径集，是一个 list。

11.3.2 platform 模块

python 中，platform 模块给我们提供了很多方法去获取操作系统的信息

【例 11.3】

```
import platform

#打印当前操作系统类型
print("操作系统类型：",platform.system());

#打印当前操作系统的版本号
print("操作系统的版本号：",platform.version());

#打印当前操作系统名称及版本号
print("操作系统名称及版本号：",platform.platform());

#获取计算机类型信息
print("计算机类型信息：",platform. machine());

#获取当前计算机的网络名称
```

```
print("计算机的网络名称：",platform.node());

#打印当前计算机的处理器信息
print("当前计算机的处理器信息：",platform.processor());

#打印当前计算机的综合信息
print("当前计算机的综合信息：",platform.uname());

#打印 Python 版本信息
print("Python 版本信息：",platform.python_build());

#打印 Python 主版本号
print("打印 Python 主版本信息：",platform.python_version());
print(platform.python_version_tuple());

#打印 Python 的编译器信息
print("Python 的编译器信息：",platform.python_compiler());

#获取 Python 分支信息
print("Python 分支信息：",platform.python_branch());

#获取 Python 解释器的实现版本信息
print("获取 Python 解释器的实现版本信息：",platform. python_implementation());
```

运行结果：

```
操作系统类型： Windows
操作系统的版本号： 10.0.17134
操作系统名称及版本号： Windows-10-10.0.17134-SP0
计算机类型信息： AMD64
计算机的网络名称： DESKTOP-QMM8CL3
当前计算机的处理器信息： Intel64 Family 6 Model 158 Stepping 9, GenuineIntel
当前计算机的综合信息： uname_result(system='Windows', node='DESKTOP-QMM8CL3', release='10', version='10.0.17134', machine='AMD64', processor='Intel64 Family 6 Model 158 Stepping 9, GenuineIntel')
Python 版本信息： ('v3.7.3:ef4ec6ed12', 'Mar 25 2019 22:22:05')
打印 Python 主版本信息： 3.7.3
('3', '7', '3')
Python 的编译器信息： MSC v.1916 64 bit (AMD64)
Python 分支信息： v3.7.3
获取 Python 解释器的实现版本信息： CPython
```

上述获取的信息中，Python 解释器的实现版本信息： CPython，默认的 Python 实现。脚本大多数情况下都运行在这个解释器中。

CPython 是官方的 python 解释器，完全按照 Python 的规格和语言定义来实现，所以被当作其他版本实现的参考版本。

11.3.3 与数学有关的模块

math 模块基础数学处理，可以实现基本道德数学运算。math 模块定义了 e（自然对数）和 pi（π）两个常量。

函数	说明	实例
math.e	自然常数 e	>>> math.e 2.718281828459045
math.pi	圆周率 pi	>>> math.pi 3.141592653589793
math.degrees(x)	弧度转度	>>> math.degrees(math.pi) 180.0
math.radians(x)	度转弧度	>>> math.radians(45) 0.7853981633974483
math.exp(x)	返回 e 的 x 次方	>>> math.exp(2) 7.38905609893065
math.expm1(x)	返回 e 的 x 次方减 1	>>> math.expm1(2) 6.38905609893065

math.log(x[, base])	返回 x 的以 base 为底的对数，base 默认为 e	>>> math.log(math.e)
		1.0
		>>> math.log(2, 10)
		0.30102999566398114

math.log10(x)	返回 x 的以 10 为底的对数	>>> math.log10(2) 0.30102999566398114
math.log1p(x)	返回 1+x 的自然对数（以 e 为底）	>>> math.log1p(math.e-1) 1.0
math.pow(x, y)	返回 x 的 y 次方	>>> math.pow(5,3) 125.0
math.sqrt(x)	返回 x 的平方根	>>> math.sqrt(3) 1.7320508075688772
math.ceil(x)	返回不小于 x 的整数	>>> math.ceil(5.2) 6.0
math.floor(x)	返回不大于 x 的整数	>>> math.floor(5.8) 5.0
math.trunc(x)	返回 x 的整数部分	>>> math.trunc(5.8) 5

math.modf(x)	返回 x 的小数和整数	>>> math.modf(5.2) (0.20000000000000018, 5.0)
math.fabs(x)	返回 x 的绝对值	>>> math.fabs(-5) 5.0
math.fmod(x, y)	返回 x%y（取余）	>>> math.fmod(5,2) 1.0

math.fsum([x, y, ...])	返回无损精度的和	>>> 0.1+0.2+0.3
		0.6000000000000001
		>>> math.fsum([0.1, 0.2, 0.3])
		0.6

math.factorial(x)	返回 x 的阶乘	>>> math.factorial(5) 120

math.isinf(x)	若 x 为无穷大，返回 True；否则，返回 False	>>> math.isinf(1.0e+308)
		False
		>>> math.isinf(1.0e+309)
		True

math.isnan(x)	若 x 不是数字，返回 True；否则，返回 False	>>> math.isnan(1.2e3) False
math.hypot(x, y)	返回以 x 和 y 为直角边的斜边长	>>> math.hypot(3,4) 5.0
math.copysign(x, y)	若 y<0，返回-1 乘以 x 的绝对值； 否则，返回 x 的绝对值	>>> math.copysign(5.2, -1) -5.2
math.frexp(x)	返回 m 和 i，满足 m 乘以 2 的 i 次方	>>> math.frexp(3) (0.75, 2)
math.ldexp(m, i)	返回 m 乘以 2 的 i 次方	>>> math.ldexp(0.75, 2) 3.0
math.sin(x)	返回 x（弧度）的三角正弦值	>>> math.sin(math.radians(30)) 0.49999999999999994
math.asin(x)	返回 x 的反三角正弦值	>>> math.asin(0.5) 0.5235987755982989
math.cos(x)	返回 x（弧度）的三角余弦值	>>> math.cos(math.radians(45)) 0.7071067811865476
math.acos(x)	返回 x 的反三角余弦值	>>> math.acos(math.sqrt(2)/2) 0.7853981633974483
math.tan(x)	返回 x（弧度）的三角正切值	>>> math.tan(math.radians(60)) 1.7320508075688767
math.atan(x)	返回 x 的反三角正切值	>>> math.atan(1.7320508075688767) 1.0471975511965976
math.atan2(x, y)	返回 x/y 的反三角正切值	>>> math.atan2(2,1)

		1.1071487177940904
math.sinh(x)	返回 x 的双曲正弦函数	
math.asinh(x)	返回 x 的反双曲正弦函数	
math.cosh(x)	返回 x 的双曲余弦函数	
math.acosh(x)	返回 x 的反双曲余弦函数	
math.tanh(x)	返回 x 的双曲正切函数	
math.atanh(x)	返回 x 的反双曲正切函数	
math.erf(x)	返回 x 的误差函数	
math.erfc(x)	返回 x 的余误差函数	
math.gamma(x)	返回 x 的伽玛函数	
math.lgamma(x)	返回 x 的绝对值的自然对数的伽玛函数	

【例 11.4】

```
import math
print(math.e);
print(math.pi);
```

运行结果如下：

```
2.718281828459045
3.141592653589793
```

random 模块

方法	原型	具体说明
random()	random.random()	生成一个 0 到 1 的随机浮点数: 0 <= n < 1.0
uniform	random.uniform(a, b)	用于生成一个指定范围内的随机符点数，两个参数其中一个是上限，另一个是下限。如果 a > b，则生成的随机数 n 满足 a <= n <= b。如果 a <b， 则 b <= n <= a
randint	random.randint(a, b)	用于生成一个指定范围内的整数。其中参数 a 是下限，参数 b 是上限，生成的随机数 n: a <= n <= b

randrange	random.randrange ([start], stop[, step]) (y,x)	从指定范围内，按指定基数递增的集合中 获取一个随机数。如：random.randrange(1, 10, 2)，结果相当于从[2, 4, 6, 8]序列中获取一个随机数
choice	random.choice (sequence)	从序列中获取一个随机元素。参数 sequence 表示一个有序类型，可以是列表、元祖或字符串
shuffle	random.shuffle (x[, random])	用于将一个列表中的元素打乱。x 是一个列表
sample	random.sample(sequence, k)	从指定序列中随机获取指定长度（k）的片断。原有序列不会被修改

【例 11.5】随机生成一个 0~100 的整数

```
import random
print(random.randint(0,99))
```

运行结果

```
7（此结果每次运行都是随机）
```

【例 11.6】将一个列表中的元素打乱

```
import random
list = [1, 2, 3, 4, 5, 6]
random.shuffle(list)
print(list)
```

运行结果：（下列结果每次运行都是随机）

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

fractions 模块，该用于表现和处理分数。

【例 11.7】分数显示

```
import fractions
x = fractions.Fraction(1, 3)
```

```
print(x)
```

运行结果

```
1/3
```

【例 11.8】随机生成一注双色球号码 - - (要求同色号码不重复)

首先分析双色球规则：双色球每注投注号码由 6 个红色球号码和 1 个蓝色球号码组成。

- 红色球号码从 1—33 中选择；
- 蓝色球号码从 1—16 中选择；
- 请随机生成一注双色球号码。（要求同色号码不重复）

```
from random import randint
red_balls = []
while len(red_balls) != 6:
    red_ball = randint(1,33)
    if red_ball not in red_balls:
        red_balls.append(red_ball)

blue_ball = randint(1,16)

red_balls.sort()

print("红球：",red_balls)
print("篮球：",blue_ball)
```

运行结果：

```
篮球：  6
红球：  [1, 9, 17, 21, 28, 29]

Process finished with exit code 0
```

第 2 次运行结果;

```
红球：  [5, 11, 20, 27, 29, 30]
篮球：  6

Process finished with exit code 0
```

11.3.4 time 模块

计算机可以使用时间戳和 struct_time 数组两种方式表示时间。

【例 11.9】调用 time.time()函数可以获取当前时间的时间戳

```
import time
print(time.time())
```

运行结果：

```
1557975231.0930007
```

上述时间对于用户而言是无意义的一串数字，需要调用 time.localtime()函数，将一个时间戳转换成一个当前时区的 struct_time。

【例 11.10】使用 time.localtime()函数将时间戳转换为当前时区

```
import time
print(time.localtime(time.time()))
#将时间数据结构转换为格式字符串进行输出
print(time.strftime('%Y-%m-%d',time.localtime(time.time())))
```

运行结果：

```
time.struct_time(tm_year=2019, tm_mon=5, tm_mday=16, tm_hour=10, tm_min=59,tm_sec=26, tm_wday=3, tm_yday=136, tm_isdst=0)
2019-05-16
```

上述函数，time.strftime(格式字符串，struct_time 时间)，是将时间数据结构转换为格式字符串进行输出

格式字符串中可以使用的日期和时间符号如下：

- %y 两位数的年份表示（00-99）；
- %Y 四位数的年份表示（000-9999）；
- %m 月份（01-12）；
- %d 月内中的一天（0-31）；
- %H 24 小时制小时数（0-23）；
- %I 12 小时制小时数（01-12）；
- %M 分钟数（00=59）；

- %S 秒（00-59）；
- %a 本地简化星期名称；

【例 11.11】直接获取当前时间的字符串

```
import time
print(time.ctime())
```

运行结果：

```
Thu May 16 11:00:33 2019
```

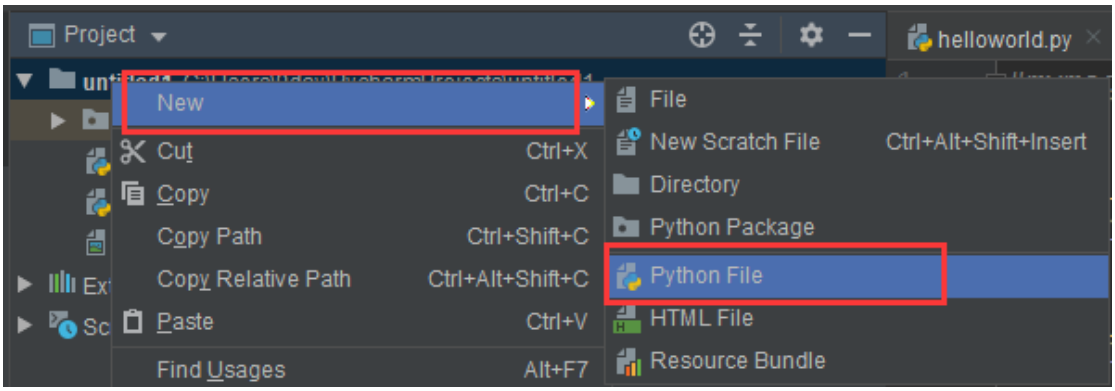
11.4 创建自定义模块

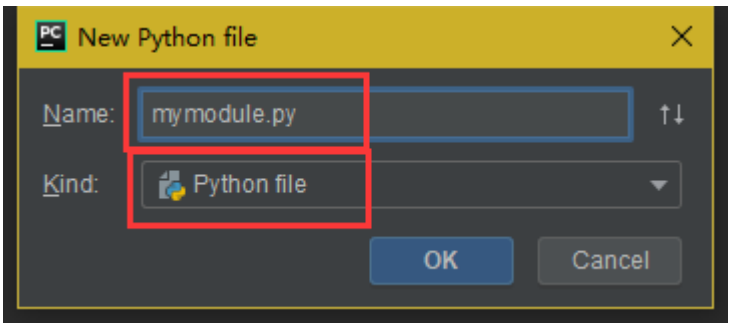
创建自定义模块，估计自己也感觉到非常有趣了，因为自己能创，也能调用其它人创建的模块使用，在工作量上面是轻松多了，这边所谓的自建模块的后缀名是(.py),还是通过示例来解说一下。特别注意一下，自己创建的模块得放到同一文件夹里面，虽然 python 有指令可以跨文件夹调用，但结果是不理想的。因为调用一下，下次这个指令将失效。

11.4.1 编写第 1 个模块

【例 11.12】创建一个模块 mymodule.py，其中包含 2 个函数 PrintString()和 sum()。

在创建自定义模块的操作步骤，在导航栏的项目名称上鼠标右键，如下图所示：





新建好后，输入如下代码，并保存：

```
#mymodule.py 模块
print("mymodule 模块");
# 打印字符串
def PrintString(str):
    print(str);
#求和
def Sum(num1, num2):
    print(num1 + num2);
```

假定【例 11.12】中创建的模块 mymodule.py 保存在与例 ex11-12.py 同一目录下，引用其中包含的函数 PrintString()和 sum()，代码如下：

```
# 导入 mymodule 模块
import mymodule
#调用 PrintString()函数
mymodule.PrintString("Hello Python")
#调用 Sum()函数
mymodule.Sum(1,2)
```

运行结果：

```
mymodule 模块
Hello Python
```

```
3
```

注意这里的运行结果，除了显示 Hello Python 和运算结果 3，print(“mymodule 模块”);语句虽然没有特意去调用，但该语句仍然被执行了一次，这如何解释呢？实际上 import 的过程就是把相应模块文件的代码执行一遍而已，也就是说对于这个例子，解释器遇到 import mymodule 后会先去执行一遍 mymodule.py，然后才回到 main.py 继续执行后边的代码。

11.4.2 用 __name__ 防止模块被错误的执行

从上一节实验可得知，模块文件实际上也是普通的 Python 源代码文件，只不过被 import 的时候当作模块来解析而不是直接执行。但在【例 11.12】中，模块中的语句如果不被定义在函数中，只要在导入模块时，该语句都将被执行，那么在代码中将无法分清执行模块的语境到底是导入的还是直接执行的？在 Python 中有一个全局变量__name__是用来确定当前模块的名称，下边我们通过一个实例来解释。

【例 11.13】对上一个示例进行部分代码调整

模块 1

```
#mymodule.py 模块

if __name__=='__main__':
    print('这个执行是从自定义模块直接启动');
    print('直接将自定义模块当作 main 启动')
elif __name__=='mymodule':
    print("这个执行是从其他模块导入 mymodule 后调用启动");
else:
    print(__name__)#其他启动的可能性

# 打印字符串
```

```
def PrintString(str):
    print(str);
#求和
def sum(num1, num2):
    print(num1 + num2);
```

主模块

```
#main.py
#  导入 mymodule 模块
import mymodule
#调用 PrintString()函数
mymodule.PrintString("Hello Python")
#调用 sum()函数
mymodule.sum(1,2)
```

直接运行 main 模块，运行结果如下：

```
这个执行是从其他模块导入 mymodule 后调用启动
Hello Python
3
```

跳开 main 主模块，使用鼠标右键运行 mymodule 模块的运行结果如下：

```
这个执行是从自定义模块直接启动
直接将自定义模块当作 main 启动
```

从运行结果上可以分析出这样一个结论，对于 Python 源代码文件，当以模块导入的形式执行它时，__name__会被设置为这个模块的名字，而对于直接将该模块作为主程序运行时，此时__name__的变量值是 mymodule。通过这一点我们可以防止一个模块被错误的执行。

11.4.3 重载模块

在上述示例中，我们了解到导入模块的过程实际就是解释器去执行一遍的过程，在某些情况下，若导入的模块代码是热修改的，我们通常需要重新导入这个模块，此时将不能静态的调用 import，只能通过 Python 内置的函数 reload 来实现模块的重载。

比如【例 11.12】示例中的 mymodule 的重载代码：

```
reload(mymodule)
```

运行结果：

```
这个执行是从其他模块导入 mymodule 后调用启动
Traceback (most recent call last):
  File "C:/Users/9day/PycharmProjects/untitled1/main.py", line 5, in <module>
    reload(mymodule)
NameError: name 'reload' is not defined

Process finished with exit code 1
```

上述运行错误是因为在 python3 中，需要从 imp 中导入。

【例 11.14】重载模块

```
#main.py
#  导入 mymodule 模块
import mymodule

from   importlib import reload # imp 从 Python 3.4 之后弃用，importlib 代替

#调用 PrintString()函数
mymodule.PrintString("Hello Python")
#调用 sum()函数
mymodule.sum(1,2)

reload(mymodule)
```

mymodule.py 仍然沿用示例 11.12 中的模块

运行结果：

```
这个执行是从其他模块导入 mymodule 后调用启动
Hello Python
3
这个执行是从其他模块导入 mymodule 后调用启动
```

11.5 程序打包

扫码看视频 11.3

python 是一个很有趣的语言，可以在命令行窗口运行。Python 有很多功能强大的模块。下边

将告诉各位如何利用 pyintaller 模块，将 Python 代码打包成 exe 文件。

视窗键+R，弹出窗口中输入 cmd，在命令行窗口中输入命令 pip install pyinstaller 后回车，几分钟后将安装成功，若提示失败，通常是因为网络问题，可再次输入上述 pip 命令。安装完毕后，进入 Python 安装目录下可找到 pyinstaller，以本书实验环境为例，路径是在 C:\Users\9day\AppData\Local\Programs\Python\Python37\Scripts，其中 9day 是编者所使用电脑 win10 的账户名称。也可以通过 pip 命令来查看安装路径 pip show pyinstaller。

11.5.1 Pyinstaller 的使用

进入需要打包的目录下，执行打包命令

Pyinstaller [opts] yourprogram.py

例如：

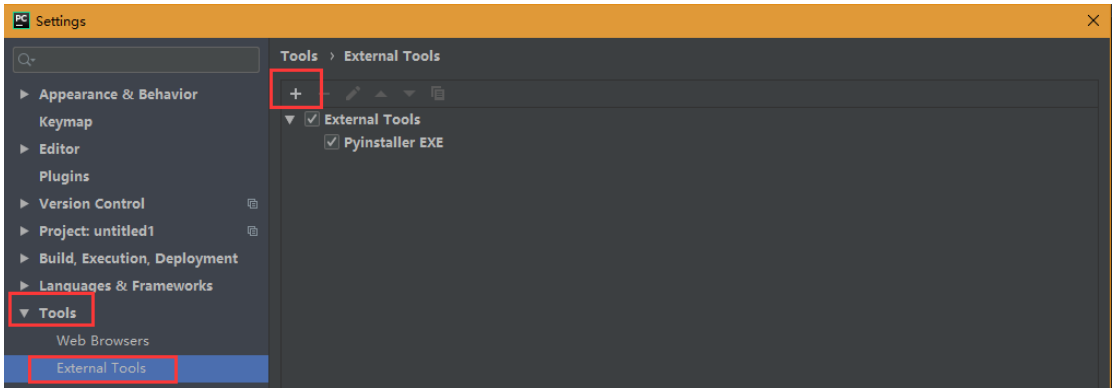
```
pyinstaller -F -W helloworld.py
```

opts 可选的参数

参数	含义
-F	-onefile,打包成一个 exe 文件
-D	-onefile，创建一个目录，包含 exe 文件，但会依赖很多文件（默认选项）
-c	-console，-nowindowed，使用控制台，无窗口（默认）
-w	-Windowed，-noconsole，使用窗口，无控制台

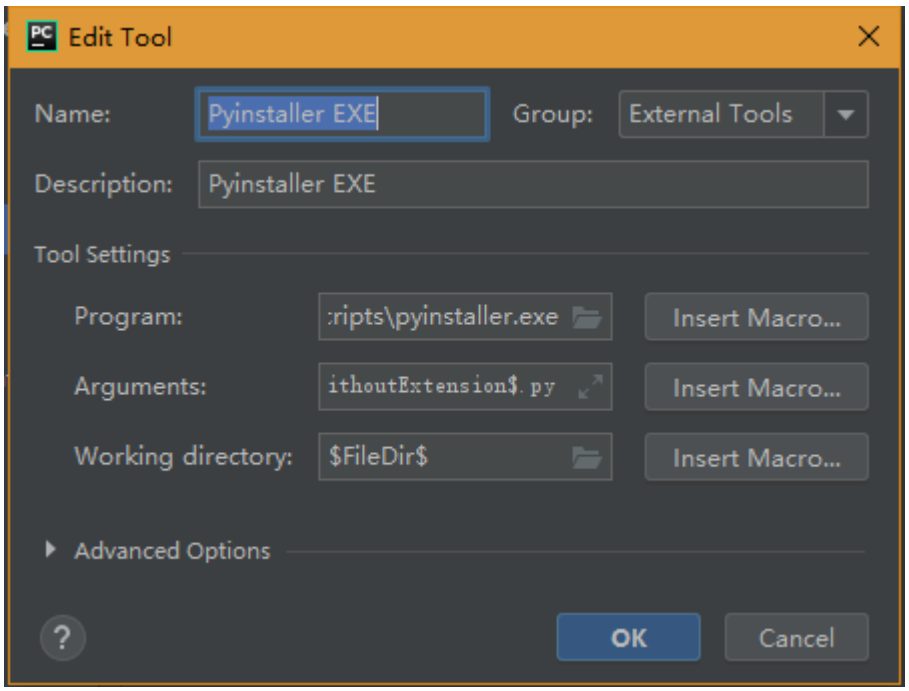
11.5.2 Pyinstaller 在 pycharm 的配置

在 PyCharm 界面下，使用快捷键 Ctrl+Alt+S，弹出如下界面，点击左上角的加号。

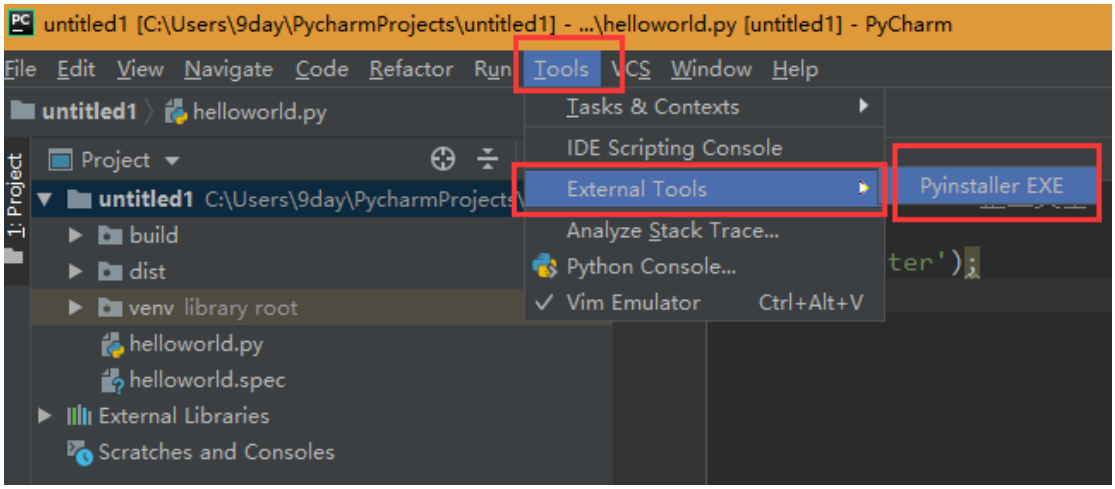


下图的属性配置具体意义

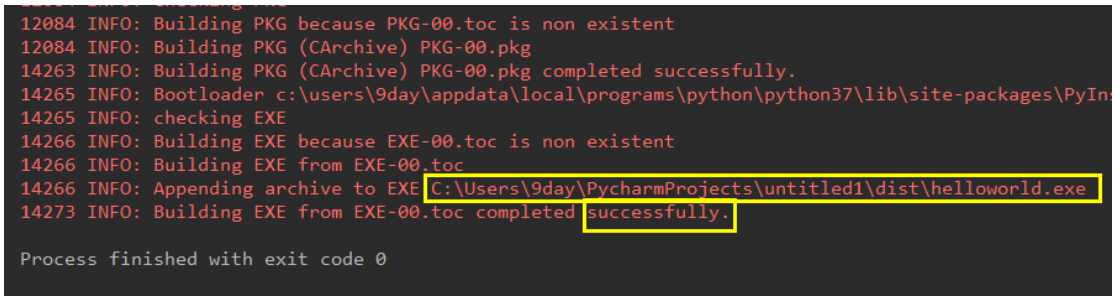
name：Pyinstaller EXE（可自定义）
Program：pyinstaller.exe 文件所在的路径，可参考下列路径
（ C:\Users\9day\AppData\Local\Programs\Python\Python37\Scripts\pyinstaller.exe）
arfuments：-F $FileNameWithoutExtension$.py
working：$FileDir$



保存设置，退出后，在当前工程界面下，选择 Tool->External Tools->Pyinstall EXE 后将进入自动编译状态，如下图所示：



编译好的 exe 文件将存储在如图所示的编译提示中的路径下。



第十二单元：用 Python 玩微信

在上一单元中我们了解到 Python 中已经集成了标准库包，同时网络上还有很丰富的第三方库模块，这一章我们将介绍一款优美的封装了微信 WebAPI 的库 wxpy，通过这个库来实现对微信账号的信息进行分析、统计、聊天等功能。

12.1 wxpy 模块概述

wxpy 这个库从官网上（https://github.com/youfou/wxpy）了解到，它是基于 itchat，使用了 Web 微信的通讯协议，通过了大量接口优化，使得模块的具备易用性和丰富的功能扩展性。具体可实现微信登录、收发消息、搜索好友、数据统计、微信公众号、微信好友、微信群基本信

扫码看视频 12.1

息获取等功能。 可用来实现各种微信个人号的自动化操作。

12.2 基本用法

【例 12.1】初始化、登录并向自己的文件助手发送一条信息

```
from wxpy import *

# 初始化机器人，扫码登陆
bot = Bot()

# 机器人账号自身
myself = bot.self

# 向文件传输助手发送消息
bot.file_helper.send('Hello from wxpy!')
```

运行结果：

```
from wxpy import *
ModuleNotFoundError: No module named 'wxpy'
```

出现这个错误主要是没有安装这个第 3 方库，解决办法根据官方手册可使用 pip 命令完成，但本书实验基于的 IDE（PyCharm）可以通过 IDE 的图形化界面进行安装。两种方法具体如下：

方法 1：（官方介绍）

从 PYPI 官方源下载安装 (在国内可能比较慢或不稳定):

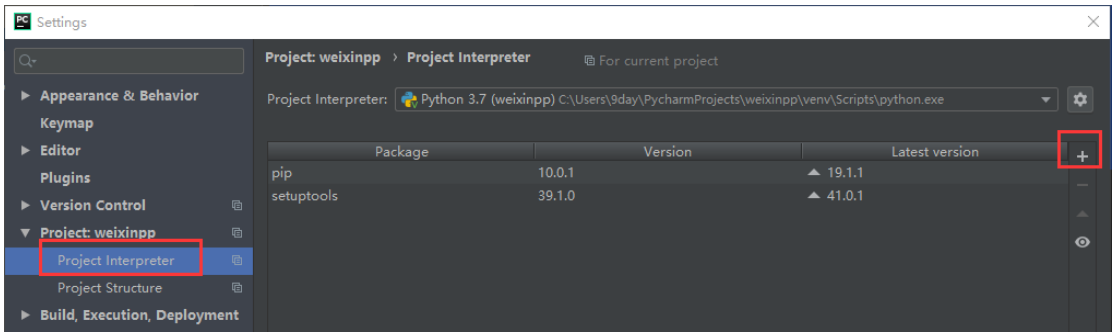
```
pip install -U wxpy
```

从豆瓣 PYPI 镜像源下载安装 (**推荐国内用户选用**):

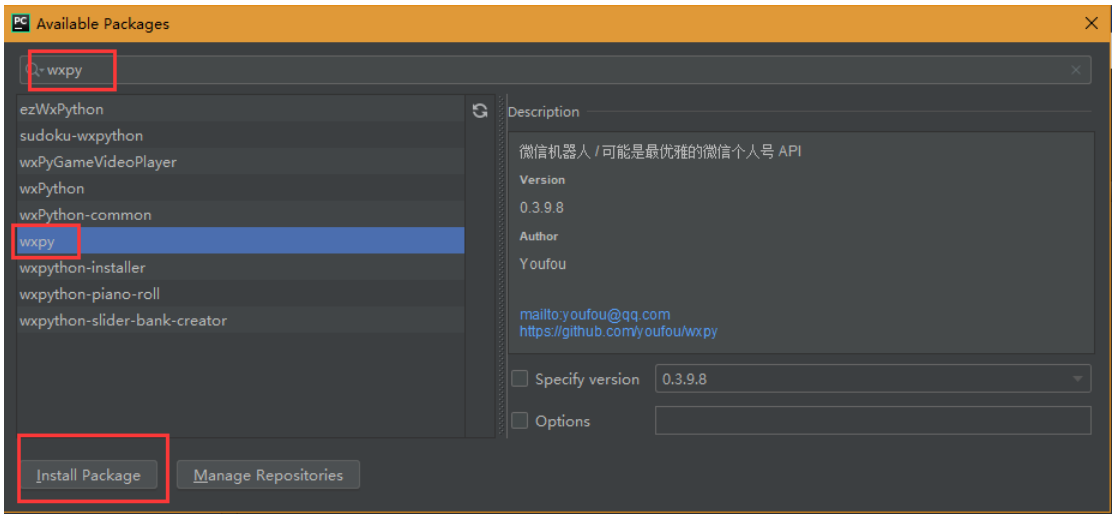
```
pip install -U wxpy -i "https://pypi.doubanio.com/simple/"
```

方法 2：

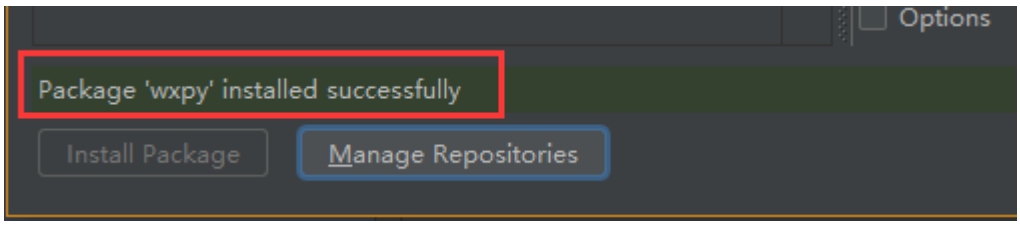
在 PyCharm 界面下，使用快捷键 Ctrl+Alt+S，弹出如下界面，选择本章建立的 weixinapp 项目



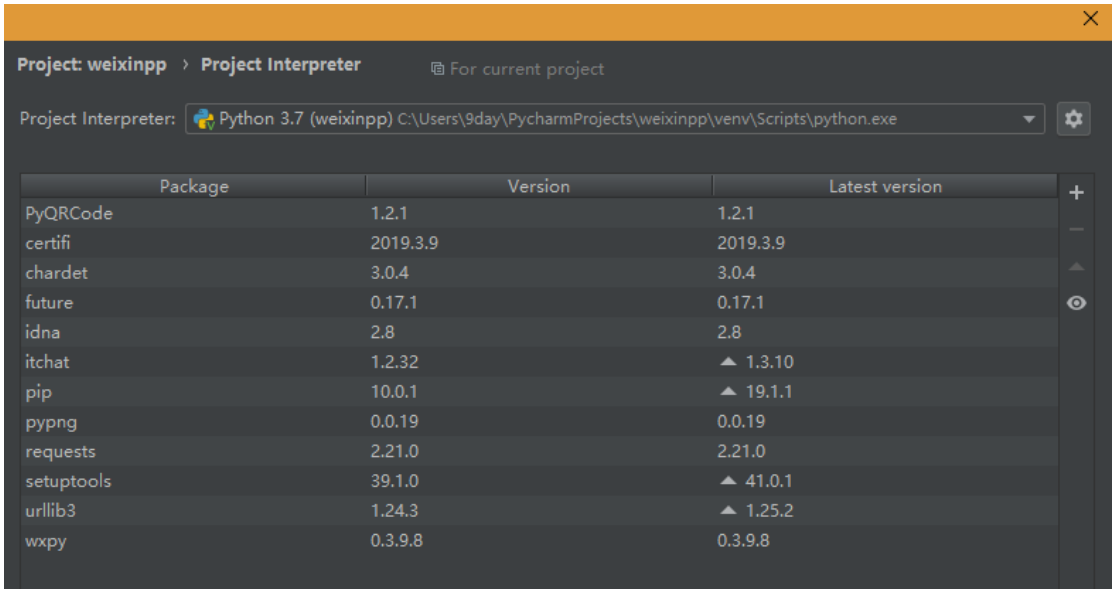
在上述界面点击右上角“+”后，将弹出如下界面



在界面的搜索栏输入“wxpy”将会出现筛选好的列表，选择 wxpy，点击 Install Package 按钮，等候几分钟后出现如下界面



当出现上述界面后，表示成功安装，此时可关闭上述窗口，项目窗口中将会增加很多模块，这些模块是在安装 wxpy 模块时根据依赖关系自动增加进来的，如下图所示：



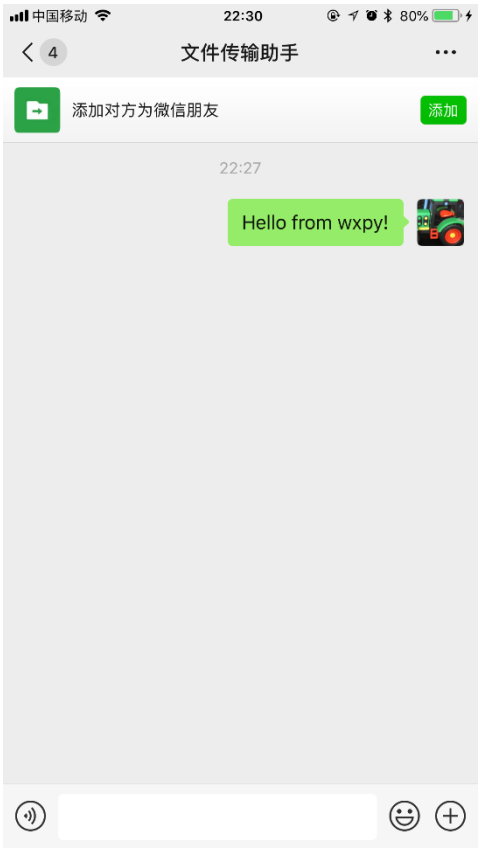
再关闭上述窗口，重新运行【例 12.1】，运行结果如下，并且会同时弹出一个二维码，此二维码需要用手机进行扫码登录。

```
Getting uuid of QR code.
Downloading QR code.
Please scan the QR code to log in.
```

在扫码后，程序运行将返回如下数据：

```
Getting uuid of QR code.
Downloading QR code.
Please scan the QR code to log in.
Please press confirm on your phone.
Loading the contact, this may take a little while.
Login successfully as  微信昵称
```

打开刚才扫码的手机，查看文件助手的消息如图



12.3 统计微信好友数、微信群、公众号

通过【例 12.1】我们发现在扫码登录向文件助手发送信息后，wxpy 机器人就自动退出了程序，若要继续实现其它功能，必须再次扫码，因此需要调用相应函数或者方法来让 bot 保持运行：

```
# 进入 Python 命令行、让程序保持运行
embed()

# 或者仅仅堵塞线程
# bot.join()
```

下边我们将通过模块提供的几个方法来获取自己的微信好友数、活跃微信群数、关注微信公众号数，具体代码如下：

【例 12.2】统计自己的微信好友数等信息

```
from wxpy import *

# 初始化机器人，扫码登陆
```

```
bot = Bot()

# 机器人账号自身
myself = bot.self

# 获取所有好友[返回列表包含 Chats 对象(你的所有好友，包括自己)]
t0 = bot.friends(update=False)
# 查看自己好友数(除开自己)
print("我的好友数："+str(len(t0)-1))

# 获取所有微信群[返回列表包含 Groups 对象]
t1 = bot.groups(update=False)
# 查看微信群数(活跃的)
print("我的微信群聊数："+str(len(t1)))

# 获取所有关注的微信公众号[返回列表包含 Chats 对象]
t2 = bot.mps(update=False)
# 查看关注的微信公众号数
print("我关注的微信公众号数："+str(len(t2)))

# 进入 Python 命令行、让程序保持运行
#embed()

# 或者仅仅堵塞线程
# bot.join()
```

运行结果：

```
Getting uuid of QR code.
Downloading QR code.
Please scan the QR code to log in.
Please press confirm on your phone.
Loading the contact, this may take a little while.
Login successfully as Cici
我的好友数：3
我的微信群聊数：0
我关注的微信公众号数：2
```

12.4 分析好友男女比例

【例 12.3】通过 bot.friends()将微信好友信息进行男女判断分析好友性别，我们首先要获得所有好友的

扫码看视频 12.2

性别信息，这里我们将每一个好友信息的 Sex 字段提取出来，然后分别统计出 Male、Female 和 other 的数目，我们将这三个数值组装到一个列表中，即可使用 matplotlib 模块绘制出饼图来

```
from wxpy import *

# 初始化一个机器人对象
# cache_path 缓存路径，给定值为第一次登录生成的缓存文件路径
bot = Bot(cache_path=r"C:\Users\9day\PycharmProjects\weixinpp\temp\wxpy.pkl")

# 机器人账号自身
myself = bot.self

friends = bot.friends(update=False)     #获取更新好友列表

male = female = other = 0

for i in friends[1:]:        #[1:]是因为整个好友列表里面自己市在第一个，排除掉
    sex = i.sex
    if sex == 1:
        male += 1
    elif sex == 2:
        female += 1
    else:
        other += 1
total = len(friends[1:])     #计算总数

#打印分析结果
print('男性好友数：',male)
print('女性好友数：',female)
print('其他好友数量：',other)
```

运行结果：

```
Getting uuid of QR code.
Downloading QR code.
Please scan the QR code to log in.
Please press confirm on your phone.
Loading the contact, this may take a little while.
Login successfully as 测试账号
男性好友数：  1
女性好友数：  1
其他好友数量：  1
```

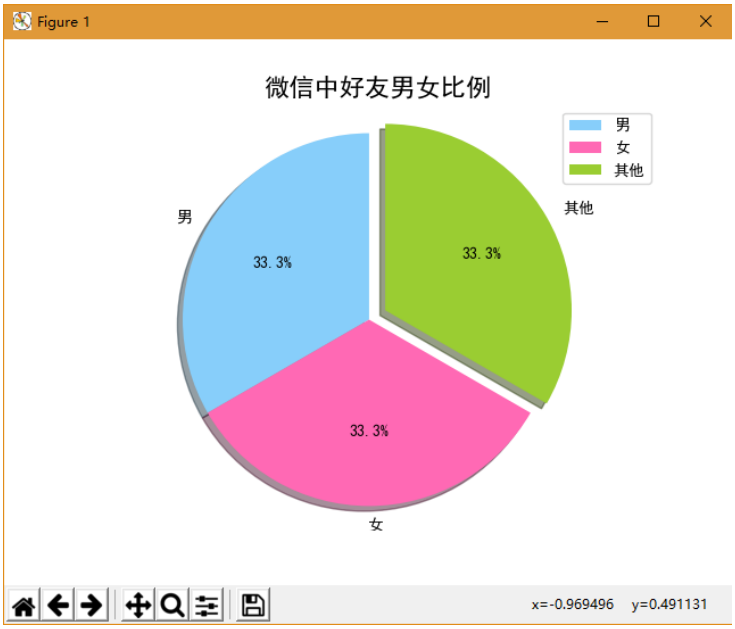
上述示例通过机器人计算出男女好友数量，如果希望统计出来的数据能更直观的显示，可以引入 matplotlib 库，以饼图方式显示。

【例 12.4】饼图显示微信中男女好友比例

```
from wxpy import *
import matplotlib.pyplot as plt

# 初始化一个机器人对象
# cache_path 缓存路径，给定值为第一次登录生成的缓存文件路径
bot = Bot(cache_path=r"C:\Users\9day\PycharmProjects\weixinpp\temp\wxpy.pkl")

# 机器人账号自身
myself = bot.self

friends = bot.friends(update=False)    #获取更新好友列表

male = female = other = 0

for i in friends[1:]:        #[1:]是因为整个好友列表里面自己市在第一个，排除掉
    sex = i.sex
    if sex == 1:
        male += 1
    elif sex == 2:
        female += 1
    else:
        other += 1
total = len(friends[1:])    #计算总数

#embed()


# 以饼图形式表现分析结果
# 切片将按逆时针方向排列和绘制

labels = '男', '女', '其他'#设置饼图每个切片的标签
sizes = [male, female, other]#设置饼图每个切片的数值
colors = ['lightskyblue', 'hotpink', 'yellowgreen']#设置饼图每个切片的颜色
explode = (0, 0, 0.1)  #用来指定每部分的偏移量

plt.rcParams['font.sans-serif'] = ['SimHei']#让 Matplotlib 显示中文
plt.title('微信中好友男女比例', size=16)#窗口标题
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
```

```
            autopct='%1.1f%%', shadow=True, startangle=90)

# 将纵横比设置为相等，以便饼图是一个圆形
plt.axis('equal')
plt.legend(loc='upper right')#显示图例
#plt.savefig('D:\\pie.png')
plt.show()
```

运行结果：



上图更直观的显示出微信好友中男女比例，对于 matplotlib 库下载与安装参考【例 12.1】中的方法即可，对于 matplotlib 库的更多功能，比如饼图切块的颜色值 colors 参数、shadow 阴影等参数，有兴趣的朋友可以在网络上搜索 matplotlib.pyplot.pie 函数的用法。

12.5 给指定朋友发送消息

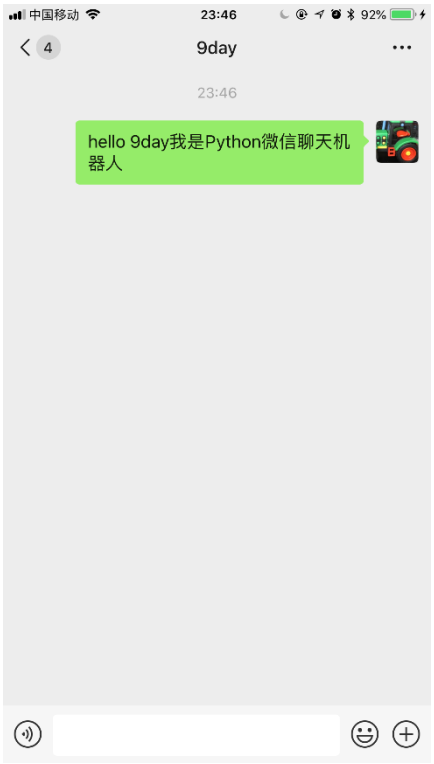
通过 bot.friends().search 方法找到好友后，可通过 send()方法发送文本，send_image()发送图片，send_video()发送视频，send_file()发送文件。

【例 12.5】给指定朋友发文字消息

```
from wxpy import *
```

```
# 初始化一个机器人对象
# cache_path 缓存路径，给定值为第一次登录生成的缓存文件路径
bot = Bot(cache_path=r"C:\Users\9day\PycharmProjects\weixinpp\temp\wxpy.pkl")

# 机器人账号自身
myself = bot.self

# 查找朋友"9day"
my_friend = bot.friends().search('9day')[0]

# 发送消息
my_friend.send('hello 9day 我是 Python 微信聊天机器人')
```

上述代码中，在初始化机器人对象时，指定了一个缓存路径，这里的目的是为了让程序运行结束后，再次运行时不用再次扫码就能直接继续运行微信机器人。但注意参数中在字符串处加了一个字母“r”，加上 r 后变为原始字符串，则不会对字符串中的"\t"、"\r"，进行字符串转义。



12.6 关键词聊天机器人

在【例 12.5】中，我们演示了发送消息给好友的效果，可以看到

消息被成功的发送给了指定的好友，但如果不是主动发消息给好友，而是机器人在被动接收到好友发来的消息后返回响应的消息回去，那么我们就需要在代码中注册一个函数来响应和处理好友发来的消息。

注册函数用@bot.register(my_friend)，my_friend 是机器人查找到的好友变量，表示注册一个响应该好友发来消息的函数，然后接下来在马上定义该函数的实现内容。

【例 12.6】关键词聊天机器人

```
from wxpy import *
import random

# 初始化一个机器人对象
# cache_path 缓存路径，给定值为第一次登录生成的缓存文件路径
bot = Bot(cache_path=r"C:\Users\9day\PycharmProjects\weixinpp\temp\wxpy.pkl")

# 机器人账号自身
myself = bot.self

# 查找聊天对象
my_friend = bot.friends().search('测试好友')[0]
#my_friend.send('hello 9day')

# 自动回复
# 如果想对所有好友实现机器人回复把参数 my_friend 改成 chats = [Friend]
@bot.register(my_friend)
def my_friednd_message(msg):
    print('[接收]' + str(msg))
    if msg.type != 'Text':    # 除文字外其他消息回复内容
        ret = random.choice(('你给我看了什么！[撇嘴]','发点文字好吗？','你真的觉得发图片好吗？[鄙视]','本 AI 还在进步中[捂脸]'))
    elif "你来自哪里" in str(msg):    # 特定问题回答
        ret = "我来自 Python 的 wxpy 机器人"
    else:            # 文字消息自动回答
        ret = random.choice(('本 AI 还在语义学习中[捂脸]','我的字典里没有这样的对话 [白眼]','反正没找到合适的话回复您[撇嘴]','你可以问我：你来自哪里[捂脸]'))
    print('[发送]' + str(ret))
    return ret
# 进入交互式的 Python 命令行界面，并堵塞当前线程
```

```
embed()
```

运行结果：

```
>>> [接收]测试好友 ：你是谁? (Text)
[发送]你可以问我：你来自哪里[捂脸]
[接收]测试好友 :(Picture)
[发送]发点文字好吗？
[接收]测试好友 ：你来自哪里 (Text)
[发送]我来自 Python 的 wxpy 机器人
```

12.7 基于图灵机器人的微信聊天机器人

事前准备 点击这里注册图灵机器人账号,然后创建一个机器人，即可获得属于你的图灵机器人 api，Tuling 库以作为 wxpy 库扩展包括在内了，不需要再另行下载和引用此库，若读者需要实验，可到图灵机器人官网 http://www.tuling123.com 注册一个账号获取一个 key。

【例 12.7】使用 Tuling 类创建属于自己的聊天机器人

```
from wxpy import *

# 初始化一个机器人对象
# cache_path 缓存路径，给定值为第一次登录生成的缓存文件路径
bot = Bot(cache_path=r"C:\Users\9day\PycharmProjects\weixinpp\temp\wxpy.pkl")

# 机器人账号自身
myself = bot.self

# 查找聊天对象
my_friend = bot.friends().search('测试好友')[0]


tuling = Tuling(api_key='此处是编者的 key，读者可单独获取***226de3ee37')
print('图灵机器人已经启动')

# 如果想对所有好友实现机器人回复把参数 my_friend 改成 chats = [Friend]
# 使用图灵机器人自动与指定好友聊天
@bot.register(my_friend)
def reply_my_friend(msg):
    ret = tuling.do_reply(msg)
    print('[发送]' + str(ret))

# 进入交互式的 Python 命令行界面，并堵塞当前线程
embed()
```

下图是运行结果，左边是 AI 机器人自动回复的：

图灵机器人可以实现查询天气、车票、翻译、基本聊天等功能，与【例 12.6】相比，图灵机器人的词库已经很完善也很全面。

小结

本单元介绍了如何利用 wxpy 第 3 方库来操作微信的方法，wxpy 还可以实现关于微信的其他功能，如果再结合 Python 的其他第 3 方库比如像图表类、数据分析类的库将能做出一个更加有趣的微信辅助系统，比如通过微信来获取家中的温湿度、开关灯等等，这里仅仅是抛砖引玉，在 Python 的世界要实现复杂的数据处理不再是难事。