Pithon

南京大学　电子科学与工程学院

撰稿人：方元

2018 年 9 月

目录

1 树莓派

1.1 Linux 简介

图 1: Linus Torvalds

Linux是一种遵循 GPL 版权协议的操作系统。它的第一作者是芬兰程序员 Linus Torvalds。在赫尔辛基大学学习计算机操作系统课程时，出于兴趣，他在 1991 年 4 月向互联网发布了 Linux 的 0.01 版，迅速受到了全世界程序员的热烈响应，自此一发不可收拾，直至发展到今天，成为有超级影响力的操作系统。成千上万的的程序员为 Linux 贡献了代码。仅 Linux 内核的维护人员名单就长达四千多行，更不要说大量的应用软件开发人员了。

Linux 社区非常活跃，差不多每 6 个月就会发布一个新的内核版本。最新版本 Linux 4.18 于 2018 年 8 月发布。Linux 是自由软件 (英语 free software，取其 “自由” 之意)。由于 free 还有 “免费” 的意思，故而也有人解释为免费软件，但自由软件的开发人员特别强调了自由和免费的区别。由于版权协议的规定，Linux 操作系统中几乎所有的软件都可以免费地获得源代码，任何人在遵守版权协议的基础上都可以对源代码进行修改和再发布，从而导致用户可以免费地获得和使用 Linux 软件这样的结果，但并不意味着开发者不能从中收取费用。

几乎所有的超级计算机都使用 Linux 操作系统[1]。在手机和平板电脑这类移动设备上，Android 操作系统占 60% 以上的市场份额 [2]，超过 iPhone 的 iOS 操作系统。

Linux 内核支持数十种处理器架构，它也是树莓派等卡片式计算机操作系统的首选。

1.2 树莓派简介

树莓派是英国一个非盈利机构树莓派基金会开发的一种卡片式计算机，最初的目的是用于对少年儿童进行计算机普及教育。第一代树莓派产品发布于 2012 年 2 月。目前最新的版本树莓派 3-B+ 于 2018 年 3 月 14 日 (这一天被叫做 π 日) 发布，它采用博通公司的处理器芯片 BCM2837 作为核心处理器，内核是 Cortex-A53 架构，是一个 4 核的 64 位 ARM处理器，主频可以达到 1.4GHz，此外还带有 VideoCore-IV 的图像处理单元 GPU。板上支持无线网络和蓝牙，并可以通过 HDMI 接口输出高清视频。

由于树莓派的成功，其他一些计算机开发商也仿照树莓派开发了类似的卡片式计算机产品。

树莓派这类计算机结构简单、体积小、耗电低，却拥有与普通计算机几乎相同的功能和性能，可以很方便地植入各种应用系统中。这种具有计算机的基本结构但又不具备普通计算机形态的计算机，业界称为 “嵌入式计算机”。所谓的 “嵌入式” 是指它是嵌在产品中的，是面向产品的专用计算机。人们看到的只是产品本身，看不到计算机。目前 98% 的计算机产品都属于嵌入式计算机。家用冰箱、微波炉、洗衣机、空调都有它们的存在。

[1]在 www.top500.org 网站公布的全球最强大的超级计算机前 500 强中，2017 年榜单上有 498 台使用 Linux，而在 2018 年 6 月的榜单上，这个数字变成了 500。

[2]Android 使用 Linux 内核。由于版权协议和一些技术细节方面的原因，Android 操作系统未被归入 Linux 的发行版

1.3 系统安装

大多数树莓派应用都安装了 Linux 操作系统。树莓派没有板载的非易失性存储设备，需要使用 microSD (Secure Digital，又称 TF 卡，TransFlash) 存储卡作为其系统运行和存储设备。它通常被划分成两个或两个以上的分区，其中第一个分区必须格式化成 FAT (File Allocation Table) 文件系统，作为 Boot 分区。该分区安装了系统引导程序 BootLoader (又称固件，firmware。树莓派的这款 firmware 不是开源软件) 和内核镜像文件，此外还有可能安装一个作为 Linux 系统初始化的 ramdisk镜像。Linux 系统正常启动后，会将根文件系统切换到另一个分区，这个分区必须是符合 Linux 根文件系统要求的格式 (EXT2/3/4FS 文件系统、REISERFS 文件系统、YAFFS 文件系统等等)。

树莓派的 BootLoader 可在 https://github.com/raspberrypi/firmware 下载。将下载的 boot 目录下的文件直接复制到 TF 卡的第一个分区即可。针对树莓派移植的内核源码在 https://github.com/raspberrypi/linux，可根据需要自行剪裁和编译，将编译后的 zImage文件替换 TF 卡 Boot 分区里的 kernel.img，就成为更新后的系统。

树莓派官网 https://www.raspberrypi.org 提供了若干 Linux 发行版。一个完整的发行版，使用方法与 PC 上的 Linux 桌面系统完全一样。

1.4 Linux 使用

目前 Linux 的桌面发行版包含了完善的图形界面环境，它的组成形式与 Windows 系统基本相同，有的只是操作习惯的差别。作为普通的计算机用户，鼠标可以完成绝大部分的日常工作。但作为程序开发人员，仅使用鼠标显然是远远不够的。

Linux 系统提供了功能强大的字符操作界面，习惯上把这种界面称为命令行方式。命令行方式需要掌握一些基本操作方法，比直接使用图形界面操作复杂，由此给人产生了 “Linux 系统难用” 这样的印象。这里需要分清普通计算机用户和计算机开发人员。显然，开发人员仅使用鼠标是远远不够的，而命令行方式则为开发人员提供了更广泛的操作空间，使得很多在图形界面下不方便甚至不可能的任务在这里得以顺利完成。

命令行界面通常由一个终端提供。终端提示符 “$” 或 “#” 下面键入的一行字符则构成了一条命令。“$” 是普通用户权限的提示符，“#” 是超级用户的提示符。一些改变系统设置的命令，普通用户的执行会受到一定的限制。

一个完整的命令行由命令、选项、参数三部分构成。命令表示你要进行何种操作，选项表示如何操作，参数表示这条命令的操作对象是什么。例如，要强制删除一个名为 hello.py 的文件：

```
$ rm -f hello.py
```

删除文件的命令是 rm，强制删除的选项是 -f (选项前面通常有一个或两个 “-”)，删除的对象是 hello.py。一些命令的选项和参数可以有多个，命令、选项、参数之间用空格分隔。

Linux 命令行中的所有字母都严格区分大小写。

下面是一些最常用的命令：

命令	功能	常用选项
ls	列文件、目录清单	-l
more	分屏打印文件内容	
cp	复制文件	-r,-a
cd	改变工作目录	
mv	文件移动/改名	-i
mkdir	创建新目录	-p
rmdir	删除空目录	
rm	删除文件	-i, -f
ps	显示系统进程表	-a,-x,-l
pwd	显示当前工作目录	
tar	打包解压/压缩文件	-z,-x,-f,-j,-J

1.5 网络连接

作为嵌入式应用，树莓派一般不配置键盘显示器这样的人–机接口 [1]。开发人员如果需要在树莓派上开发软件，最简单的方法是通过网络连接。

多数 Linux 发行版都会启动一个 ssh (Secure SHell) 的服务，它用于提供通过网络远程登录使用计算机的功能。如果树莓派启动了 sshd (字母 “d” 表示守护进程 daemon) 并连接到局域网，我们就可以在自己的 PC 终端上使用远程登录功能：

```
$ ssh pi@192.168.2.103
pi@192.168.2.103's password:
```

首次登录时，系统会给出安全提示。一旦建立安全连接后，以后只需要在每次登录或连接时输入用户密码就可以使用了。登录成功后，会看到树莓派给出的问候界面：

```
##############################################
#           Raspberry Pi3 (aarch64)          #
#                 Desktop                    #
#                 WELCOME!                   #
##############################################
Raspberrypi:~$
```

这里假设树莓派的 IP 地址是 192.168.2.103，用户名是 pi。默认的方式下，出于安全的考虑，ssh 不允许 root 用户远程登录[2]。之后在这个终端上所有的操作都是在树莓派上进行的，除了没有图形界面以外，它与本地的操作完全相同。

[1]连接人–机接口很简单，Linux 操作系统直接支持 USB 键盘/鼠标，HDMI 输出可以直连带有 HDMI 接口的显示器。

[2]类 UNIX 系统中，拥有最高操作权限的用户被称为超级用户 (super user)，用户名为 root。

ssh 服务除了提供远程登录以外，同时还提供远程文件传输。用于文件传输的本地命令是 scp，用法是：

```
$ scp user1@192.168.2.103:file1  user2@192.168.200.170:file2
```

它表示把 IP 地址 192.168.2.103 上的文件 file1 复制到 192.168.200.170 机器上，作为 file2 保存。两台机器各自拥有 user1 和 user2 用户，复制过程中会要求输入用户的密码。如果是本地文件，则可以省去用户名和 IP 地址部分。如果复制的目标文件不想重新命名，可以省去 file2 部分，但 IP 地址后面的冒号需要保留。

1.6 远程桌面

Linux 系统的网络功能非常强大，除了上面介绍的 ssh 终端连接方式以外 (ssh 也可以实现图形化方式)，一个较为简单的方法是 VNC (Virtual Network Computing) 。服务器端 (树莓派) 启动 X-Server 和 VNC 服务后，其他机器就可以通过 VNC 客户端连接服务器的图形界面。Linux 系统常用的客户端命令是 vncviewer：

```
$ vncviewer 192.168.2.103
```

如果一切正常，服务器的桌面就展现在主机的一个窗口中。

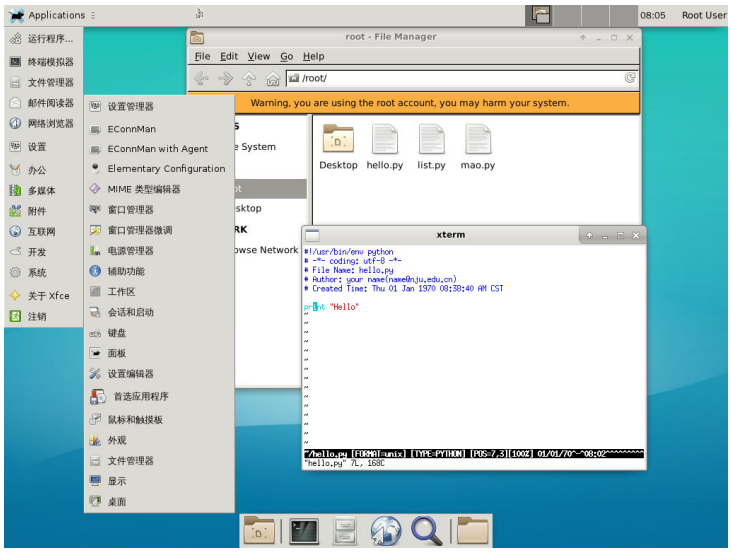


图 2: 树莓派上的 XFCE4 桌面

图2 是通过 vnc 登录到树莓派上展示的图形界面，树莓派运行 XFCE4 桌面环境。

1.7 远程网络存储

嵌入式系统通常不可能有大容量的存储空间。此时网络文件系统 NFS (Network File System) 就可以发挥作用了。使用网络文件系统时，要求 PC 端作为 NFS 的服务器，启动 NFS 服务。PC 机上的文件 /etc/exports 限定了提供网络文件服务的目录和权限：

```
/srv/nfs4     192.168.*.*(rw,sync,subtree_check,no_root_squash)
```

假设 PC 的 IP 地址是 192.168.200.170。在树莓派上执行下面的命令：

```
# mount 192.168.200.170:/srv/nfs4 /mnt -o nolock,proto=tcp
```

便可将 PC 的 /srv/nfs4 目录挂载到 /mnt 目录下。在 PC 上访问 /srv/nfs4 和在树莓派上访问 /mnt，二者访问的是同一个资源，真正的存储设备在 PC 上。mount 命令的选项 “-o” 指定的参数表示强制使用 TCP 协议 (NFS 默认优先使用 UDP 协议)。

网络连接在嵌入式开发中非常有用，它除了可以省下一套输入输出设备 (键盘、鼠标、显示器) 以外，还可以借助主机的强大的处理能力、灵活的软件配置和大量的存储空间，为嵌入式系统开发提供强有力的支持。

2 Python 基础知识

2.1 Python 简介

Python 最早出现于 1980 年代晚期[1]，它的作者是荷兰程序员 Guido van Rossum，当时就职于荷兰国家数学与计算机科学研究院 (该机构的荷兰名称缩写是 CWI)。谈及创作动机时，van Rossum 说是为了填补 1989 年圣诞节放假期间的空虚，手边正好有一台计算机，于是决定为一个新的脚本语言 (ABC 语言的继任者。ABC 语言是一种用于教学的程序设计语言) 写一个解释器。当时有一个英国的喜剧小组，名叫 Monty Python。由于 van Rossum 本人非常喜欢 Monty Python 的节目，于是他决定用 “Python” 命名他的这项工作。

Python 在英语里有 “蟒蛇” 的意思，Python 的图标就是两条纠缠在一起的蟒蛇。

图 3: Guido van Rossum

2.2 Python 发展史

Python 属于自由软件。它的版权协议几经变化，目前采用 Python 软件基金会版权协议 (Python Software Foundation License) 发布。这种版权协议是开源软件协议的一种，与通用公共版权协议 (GPL，General Public License) 兼容。与 GPL 协议不同的是，它不强制发布修改后的软件版本必须提供源代码。

目前在计算机系统中普遍使用的是 Python 2 和 Python 3 两个大版本。Python 2.0 于 2000 年 10 月 16 日发布，Python 3.0 于 2008 年 12 月 3 日发布。Python 3 并不是 Python 2 的升级版，它与先前的版本不兼容，但其中的主要特性都已经做了移植，以保持和 Python 2.6.x 和 Python 2.7.x 系列的兼容性。Python 在发展过程中，语法规则没有发生大的变化，不兼容主要

[1]官方公布的发布时间是 1991 年 2 月 20 日

体现在细节上，比如 Python 2 中，“print” 是一个关键字，而在 Python 3 中则是一个普通的函数；Python 2 的除法运算符 “/” 在进行除法运算时，结果只保留除法的整数部分，而 Python 3 则保留小数部分。2017 年发布的 Python 2.7.14 被确定是 2.7 系列的最后一个版本，Python 开发者希望逐渐淘汰 Python 2。最初 Python 2 的终止日期被定在 2015 年，但因为目前尚有大量软件在使用 Python 2，淘汰的计划不得不向后推迟。建议读者在新开发的程序中尽可能使用 Python 3，而本书也根据 Python 3 的规则讲解。

2.3 Python 的应用

Python 是多模态的编程语言，支持完整的面向对象编程和结构化编程的特性。大量扩展模块的支持，使其具有强大的功能，从数学运算、数据库处理到网络和视频处理，无不体现出其灵活和便捷。

2.4 软件环境

2.4.1 Python 编辑工具

任何编程语言都需要有一个开发环境，Python 也不例外。其实，作为像 Python 这样的脚本语言，开发环境倒真的可有可无，只要有文本编辑工具就够了。但是程序写出来后总要运行、调试吧？即使以源码形式发布的 Python 软件，安装到用户计算机上，也要求该计算机具备 Python 运行环境。

Python 官方网站 https://www.python.org/downloads 提供了各种操作系统的安装包，使用者可根据各自的需要安装相应的版本。如果愿意，也可以下载源代码在自己的机器上编译、安装。这可是一项费时费力的活儿，但有时候却不得不做，比如想在某个嵌入式平台上跑 Python。Python 是跨平台的编程语言，但运行环境与硬件平台和操作系统都有关。

Python 对 Linux 操作系统有天然的亲和力。大多数 Linux 的发行版都已默认安装了 Python 解释器，因为 Linux 的很多基础软件要依赖 Python。安装了 Python 系统后便同时具备了 Python 的运行环境和开发环境。Python 运行环境是 Python 源程序的解释器 python3 (Windows 系统则是 python3.exe) 和相关的动态链接库。如果是 Python 2，可执行程序名则是 python。Python 的集成开发环境 idle3.6 (Python 2 对应的名字是 idle) 本身就是由 Python 语言写成的。如果你有兴趣用文本编辑器打开，它仅有下面的寥寥数行：

清单 1: idle3.6

```
#! /usr/bin/python3

from idlelib.PyShell import main
if __name__ == '__main__':
    main()
```

它提供了一个运行 Python 指令的交互界面 (Python shell) 和符合 Python 编程风格的编辑工具。不得不承认，Linux 系统中的 Python 环境 idle 确实过于简陋。不过 Linux 系统的开发人

员本来也不喜欢集成开发环境，而更多的是利用功能强大的编辑器编辑源代码，特别像 Python 这样简洁的编程语言，集成开发环境对程序员来说更是累赘。良好的编辑工具有助于提高效率，减少差错。由于 Python 对源程序版面格式有一定的要求，针对 Python 语言设计的编辑工具也方便了程序的输入。有些其他的编辑工具 (如 vim、sublime 等) 通过适当的配置也可以很方便地用于编辑 Python 源程序。

即使不启动 idle，在终端上输入不带任何参数的 python 命令本身，也会进入 python 交互环境。在这个环境下，可以逐条输入 Python 代码，随时获得这行或这段代码的结果。

2.4.2 运行 Python 程序

按照惯例，学习程序语言的第一个例子总是从一句问候语开始：

清单 2: hello.py

```
print("Hello, Python!")
```

这一行很容易看懂，但似乎有点太简单了。计算机怎么知道按照 Python 的规则运行呢？

作为一个源程序，我们将含有上面一行语句的内容保存成一个文件，比如说 hello.py，它就是 Python 的源程序。在安装了 Python 的 Windows 系统中，该文件会以 Python 的图标提示用户。只需要用鼠标点击它，系统便自动调用 Python 解释器运行这个程序，打开一个窗口，打印一串字符。只是由于过程太短，随着程序的结束，打开的窗口瞬间就关闭了。

在 Linux 系统上有多种运行方式：[1]

- 在桌面环境中，使用鼠标点击该文件的图标，方式与 Windows 中的做法相同，结果和在 Windows 中看到的效果也相同。
- Linux 操作系统习惯使用终端操作。因此可以直接在终端中运行下面的命令：

  ```
  $ python3 hello.py
  ```

  程序在终端上打印一句问候语。
- 将下面的内容写在源程序文件的第一行：

  ```
  #!/usr/bin/env python3
  ```

  并通过命令 “`chmod +x hello.py`” 为该文件设置可执行属性，该文件形式上就成为一个独立的可执行文件，可以通过下面的命令直接运行：

  ```
  $ ./hello.py
  ```

  文件开头的两个字节 “#!” 是脚本文件的标识，系统会根据后面指定的程序作为解释器。

Python 一行可以有多个语句，每个语句之间用分号分割，最后一个语句后面的分号可有可无。表达式之间可以有空格，但每一行语句最前面的空格 (缩进) 是有意义的，不要随便加 (有关缩进的问题将在后面讨论)。上面这个小程序，每行代码都应该顶格书写。

[1]类 UNIX 系统，包括 UNIX、iOS 及 Linux 的各种发行版，运行方式基本相同。

2.4.3 交互方式

终端上键入 “python3” 或者 “idle &” 可以启动交互方式，在提示符 “>>>” 下可以直接输入 Python 语句：

```
Python 3.5.2 (default, Sep 14 2017, 22:51:06)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("Hello, Python!")
Hello, Python!
>>>
```

这个环境下的操作不能保存，语句写错了也不便于修改，因此不是一个真正的编程环境，但对于一些简单的操作和不太复杂的程序来说，具有反应及时，操作便捷的优点。作为基本操作，我们利用这个环境学习 Python 的一些基础知识。需要查询一些内建函数和对象的文档，可以简单地调用 help() 方法；对于非内建模块，导入后也可以使用 help() 查阅。

idle 则是一个图形化工作环境，通过 File 菜单的 “打开” 或 “新建” 子菜单，会创建一个新的编辑窗口，真正的编程工作可以在这个环境下完成。

2.5 数值表示

2.5.1 常数

Python 中，表示常数的数值类型有整数、浮点数和复数三种。书写时，没有小数点的数字被当作整数处理，数字中带有小数点的被当作浮点数处理，如果数字后面紧跟着字母 “j” 或 “J”，则这个数就是复数，字母 “j” 或 “J” 是虚数单位。以上需要注意的是：

- 在 Python 2 中，整数之间的四则运算 (主要是除法运算) 结果总是整数，因此 3/4=0。如果要提高计算精度，必须至少将其中的一个数字用浮点数表示。但 Python 3 改变了这一规则，除法运算结果总是用浮点数表示，因此 3/4=0.75。如果仍想让结果保持整数，则应使用整除运算符 “//”。整除运算的结果并不一定是整型值，只是小数部分为零。
- 虚数单位 “j” 或 “J” 必须紧跟在数字后面，中间不能写乘号 “*”，也不能写在数字前面，否则会被作为变量对待。

试看下面的操作：

```
>>> 12345678+87654321
99999999
>>> 1000/330
3.0303030303030303
>>> (1 + 2j)*(3 - 2j)
(7+4j)
```

```
>>> 2.8 // 0.03
93.0
>>> (1 + j)*(1 - j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'j' is not defined
```

缺省的数字格式被作为十进制数字处理。Python 语言中可以将数值用二进制 、八进制和十六进制表示，表示方法是在数字前面分别加上 0b、0o、0x (字母不分大小写。三个字母分别是 binary、octal 和 hexadecimal 三个单词的首字母)，例如：

```
>>> 0xff - 0b1111
240
```

2.5.2 变量

变量是用于存储信息的单元。Python 中没有单独声明变量的语句，为变量赋值无需事先定义，赋值的同时也就声明了这个变量。未经赋值的变量则被认为是不存在的对象，不能参与运算。看下面的例子：

```
>>> a1 = 10
>>> a2 = a2 + a1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a2' is not defined
>>> a2 = 20
>>> a2 = a2 + a1
>>> a1, a2
(10, 30)
```

注意，在多数计算机语言中，单个 “=” 符号不是等号，而是赋值号，它表示将该符号右边的表达式结果传递给左边的变量中。等号有专门的表示方法，我们在稍后会看到。

Python 允许同时为多个变量赋值。基于此，两个变量交换也只需要一个语句：

```
>>> a, b = 200, 'hello'
>>> b, a = a, b
>>> print ('a=%s,' % a, 'b=%d' % b)
a=hello, b=200
```

上面使用了函数 print() 的格式化打印功能，引号中的 “%” 表示打印变量的属性，而引号外面的 “%” 用于关联格式变量。表1列出了常用的打印格式。

格式字母前面可以有数字，表示格式化字符的长度，长度不足时，前面用空格或 “0” 填充：

表 1: 变量的格式化表示

格式	含义
%s	字符串
%d, %i, %u	十进制整数
%o	八进制整数
%x, %X	十六进制整数
%e, %E	科学记数格式
%f, %F	浮点数

```
>>> "%4x" % 123
'  7b'
>>> "%04X" % 456
'01C8'
```

Python 中，合法的变量名由若干个字母、数字和下划线构成，字母区分大小写，首个字符不能是数字。由于 Python 3 支持 Unicode，因此，像下面的名称也是合法的：

```
>>> 张三的年龄 = 32
>>> 李四 = 张三的年龄 + 3
>>> print(李四)
35
```

但不建议使用这种命名方式。

2.6 字符串

2.6.1 字符串的表示

变量除了可以保存数值信息，也可以保存非数值信息。字符或字符串 就是典型的非数值信息：

```
>>> greeting = "Hello, Python!"
>>> print (greeting)
Hello, Python!
>>> number = 35
>>> number + number
70
>>> string = "35"
>>> string * 3
'353535'
```

一对引号之间的内容被当作字符串处理，即使是数字，也不具备数值含义。

表示字符串的引号可以用单引号也可以用双引号，但前后不能混用。如果字符串本身含有引号，可以使用下面的方法之一处理：

- 交替使用引号。例如字符串中有双引号时，则可用单引号作为字符串的引号。
- 在字符串中的引号前面加反斜线 “\”，进行转义处理。

反斜线被称作转义符，它表示后面紧跟的一个字母或符号将改变原有的含义。转义符通常还用于处理一些不能直接用字母表示的场合，例如换行 (“\n”)、制表符 (“\t”)、续行 (“\”) 等等。

多行文本可以使用三引号 ''' 或 """。

字符串除了实现程序功能以外，也是程序注释和帮助文档的重要组成部分。

Python 中的字符串有三种类型：裸字符串 (raw string)、Unicode 编码字符串和二进制字符串 (binary string)，分别用对应的字母前缀声明。像下面的例子：

```
>>> rStr = r"Python is easy."
>>> uStr = u'Python isn\'t hard to learn.'
>>> bStr = b'Hello.!'
```

对于英文 ASCII (American Standard Coding for Information Interchange) 字符和符号，裸字符串和 Unicode 字符串是等价的 (裸字符串不接受转义符)，但对其他语言文字则不同。我们在使用中文时特别需要注意。在中国大陆，中文存在两种不同的编码方案 (中国国家标准 GB18030 和国际标准 Unicode)，他们在计算机中的信息存储方式互不兼容，一些中文显示乱码的根源就在于此。Python 建议将字符串用 Unicode 表示。因此，一个标准的 Python 源程序开头两行应该是这样的：[1]

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
```

UTF–8 是 Unicode 实现方案的一种，它使用若干个 8 位的二进制按一定的规则拼接表示一个文字符号的 Unicode。英文字符的 ASCII 码和 UTF–8 码恰好相同。使用二进制字符串定义的字符，则可以直接得到字符的 ASCII 码：

```
>>> print (bStr[0],bStr[1],bStr[2])
72 101 108
```

2.6.2 字符串的基本运算

字符串属于常量的一种，一旦设定，其中的元素便不可更改，例如：

```
>>> str = 'hello'
>>> print (str[0])
h
```

[1] “#” 在 Python 中可以作为单行注释符，但这两行的 # 比较特殊，不是注释。

```
>>> str[0] = 'H'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item assignment
```

Python 定义了字符串的加法和字符串与整数的乘法。字符串相加就是将字符串按书写顺序前后串接；字符串与整数相乘，等效于若干个字符串相加。

2.7 逻辑值

逻辑值又称布尔值 (boolean 的音译)，它只有 “真” 和 “假” 两个状态，分别用 True 和 False 表示。在其他类型的对象中，以下几种情况的逻辑值为 False：

1. False 本身；
2. 任何类型的数值 0 (整数、浮点数、复数)；
3. 长度为 0 的字符串；
4. 任何一种仅有空元素的数据结构 (空词典、0 个元素的集合、列表或元组，以及自定义的、不包含任何成员的数据结构[1])；
5. 空对象 None 。

除此以外的其他对象，如果作为逻辑值使用时，都是 True。

None 是一个较特殊的对象，它表示什么对象也不是。当一个函数不返回任何值时，调用者得到的返回值就是 None。

逻辑值常常来自条件判断，因此 “`False == 0`” 条件成立，它的结果为 True。这里连写在一起的两个等号 “==” 是表示相等的条件判断。

作为变量参与数值运算时，逻辑变量的值是 1 (True) 或 0 (False)。

2.8 函数

将一组实现特定功能的代码组织起来，形成一个可供其他代码直接调用的接口，这段代码就是函数。一个函数通常有若干个输入参数和若干个输出参数 (输出参数通常又被称为返回值)。前面用到的 print() 就是一个函数的典型例子。

再如，内建函数 pow(x, y)用于计算 x^y，它甚至可以实现复数运算功能：[2]

```
>>> pow(-1, 0.5)
(6.123233995736766e-17+1j)
```

[1] 自定义对象的逻辑值可以根据对象本身的需要决定，但遵循这个原则总是有好处的。

[2] 严格地说，-1 开平方的结果是 $\pm 1j$，同其他语言的函数功能一样，Python 的数学函数只给出主值结果。这里计算出的结果实部不等于 0, 原因是由计算精度导致的。

内建的函数 pow() 与 Python 的运算符 “**” 是等价的。

Python 的函数形式非常灵活，函数可以没有输入参数；可以有多个返回值，也可以没有返回值。即使同一个函数，输入参数和返回值的数量及格式都是可变的。

2.9 模块

Python 已经内建了大量的函数，但还是远远不够，比如我们要计算以自然数 e 为底的指数，虽然可以用 pow(2.7182818, y) 这样的形式，但毕竟不方便，而且很多情况下我们也不能记住足够精度的 e。一些我们认为的常用函数 (如三角函数) 也不在内建函数中。

Python 通过导入其他模块 (俗称 “库”) 实现更为灵活的功能。上面提到的指数函数和三角函数就在数学函数模块中，

```
>>> import math
>>> math.exp(1.0)
2.718281828459045
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/3)
0.8660254037844386
```

正是由于有丰富的模块支持，才使得 Python 实现各种功能变得相当简单。

2.10 运算符和优先级

Python 一行语句可以写很长，实际编程时，并不主张使用过于复杂的运算，可以用分行、加括号的方式明确运算的先后顺序。

编程语言中，掌握运算符的用法，准确了解各种运算的优先级，对程序的正确性至关重要。以下分类列出 Python 使用的运算符及其作用

2.10.1 算术运算

表 2 列出的是算术运算符的功能。表中，除法运算符在 Python 2 中，用于整数之间的运算

表 2: 算术运算符

符号	含义	说明
+, −	加减运算	
*, /	乘除运算	
//	整除	得到商的整数部分
%	取模	得到除法的余数
**	指数 (幂) 运算	

结果，返回商的整数部分。取模运算 “%” 在字符串运算中另有含义，见2.5.2节格式化打印的例子。“*” 和 “**” 在函数的形式参数中也不再是运算符，在函数章节会详细讨论。

2.10.2 关系运算

关系运算用于比较变量的大小关系，返回逻辑值。表3 列出了关系运算符的功能。

表 3: 关系运算符

符号	含义
==	比较两个对象是否相等
!=	比较两个对象是否不等
>	前者是否大于后者
<	前者是否小于后者
>=	前者是否大于或等于后者
<=	前者是否小于或等于后者

关系运算的总体优先级低于算术运算。关系运算中的不等于 “!=” 曾经使用过 <>，现已逐渐被淘汰，建议不再使用。

2.10.3 位操作运算

位操作运算符以二进制位为操作单位，它只能用于整数或整数之间的运算。表4 列出了位运算的符号和功能。位运算虽然是以二进制位为单位，但并不要求在编程序时用二进制书写。

表 4: 位操作运算符

符号	含义	说明
&	两个数的二进制按位与操作	
|	两个数的二进制按位或操作	
^	两数的二进制按位异或操作	
~	将二进制位按位取反	
<<	左移位操作	a << n, 将 a 左移 n 位，低位补 0
>>	右移位操作	a >> n, 将 a 右移 n 位，高位保持符号位不变

Python 在内部会自动进行二进制处理。移位操作中，移位次数必须是非负的整数。

2.10.4 赋值

除了直接赋值 “=” 以外，Python 还支持一些与运算混合的赋值方式，以便让代码更加紧凑。表5 列出了所有这类赋值符号。这类赋值语句要求赋值对象必须参与运算，并且可以成为运算式中的第一个成员。像 “`a = b - a`” 就无法用这类的赋值方式表示。

表 5: 赋值运算

符号	说明
+=	a = a + b 可以写成 a += b
−=	a = a − b 可以写成 a −= b
*=	a = a * b 可以写成 a *= b
/=	a = a / b 可以写成 a /= b
%=	a = a % b 可以写成 a %= b
//=	a = a // b 可以写成 a //= b
**=	a = a ** b 可以写成 a **= b
&=	a = a & b 可以写成 a &= b
|=	a = a | b 可以写成 a |= b
^	a = a ^ b 可以写成 a ^ = b
<<=	a = a << b 可以写成 a <<= b
>>=	a = a >> b 可以写成 a >>= b

2.10.5 逻辑运算

基本逻辑运算有三个，用关键字 “and” (与)、“or” (或)、“not” (非) 表示。前两个是二元运算 (运算符前后各有一个对象)，返回两对象的逻辑运算结果。“not” 是单运算符，它返回对象相反的逻辑结果。逻辑运算的返回值是逻辑值，参与逻辑运算的对象可以是其他数据类型。

除了以上三个基本逻辑运算以外，还有两个比较特殊的逻辑运算：

- is, 用于判断前后两个实例是不是同一个对象。它与条件判断 “==” 意义不同。这里，同一个对象指的是使用同一个存储单元。同一存储单元允许有不同的符号引用，但他们的值肯定是相等的；反之，相等的值却不一定存储在同一个位置上。试看下面的例子：

```
>>> a = 2
>>> b = a
>>> a is b
False
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> a[0] = 9
>>> print (b)
[9,2,3]
```

is 的反义运算是 is not 。

- in，用于确认一个对象是否是另一个对象的所属成员。例如：

```
>>> s = 'Hello'
>>> 'e' in s
True
```

in 后面的操作数必须是可迭代的对象，如列表、元组等。in 的反义运算是 not in。

2.10.6 运算优先级

表6按从高到低顺序列出了 Python 运算优先级。同优先级运算中，按语言书写顺序先左后右的顺序执行，赋值则是从右到左的顺序。

表 6: 运算优先级

运算符	分类
**	指数 (最高)
~	位反
*, /, %, //	乘除类
+, −	加减类
>>, <<	移位运算
&	位与运算
^ , |	位或、位异或运算
<, >, <=, >=	关系运算
==, !=	关系运算
=, +=, −=, ...	所有赋值运算
is, is not	
in, not in	
not, or, and	逻辑运算符

3 结构数据

3.1 列表

3.1.1 列表的形式

列表 (list)是一组有序数据的集合，通过顺序下标 (index)访问其中的元素。下标的序号从 0 开始，第一个元素的下标是 0，第二个元素的下标是 1，...。定义一个列表时，用 “[ ]” 将列表的元素括起来，列表元素之间用逗号分隔。通过下标访问列表中的元素时，下标也使用 “[ ]” 括起来。列表中的元素可以是不同的对象类型：[1]

[1]Python 中所有结构数据中的元素都不要求是同一类对象。

```
>>> lst = [-2, 'alpha', 3.14, [1,2,3]]
>>> print (lst[0] + lst[2])
1.14
>>> print (lst[3][2])
3
```

Python 中的下标可以是负数，它表示从最后一个元素的位置倒数的序号，即：-1 表示倒数第一个元素的位置，-2 表示倒数第二个元素的位置[1]，...。

访问列表元素时，每次可以访问多个单元，此时下标是一个序列。例如针对上面的 “lst” 列表，“`print (lst[1:-1])`” 将打印第二个元素 (下标为 1) 到倒数第二个元素 (下标为 -1 的前一个)。这里需要注意的是，不能用 -0 表示访问到最后一个元素，最后一个元素下标位置空缺即表示访问到最后一个元素，如 “`print (lst[1:])`”。下一小节 “序列” 将对此进一步说明。

3.1.2 序列

上面这种下标形式，在 Python 中被称为序列 (sequence) 。一个完整的序列表示由三个数字组成：起点、终点、步长，每个数字之间用冒号分隔。序列中只有一个数字时，即表示该数字本身；如果有两个数字，这两个数字分别表示起点和终点。有冒号存在即表示前后有数字，不论数字是否明确地写出。未写明的数字，默认起点为 0，终点为序列的边界 (第一个或者最后一个，取决于步长的方向)，步长为 1。起点值被包含在序列之内，而终点值不被包含在序列内 (未显式写出的终点值除外)。下面的例子有助于理解这些概念：

```
>>> a = range(10)      # 构造一个长度为10的序列 0,1,...9
>>> b = a[:]           # 赋值给一个新的列表
>>> c = a[-1::-1]      # 逆序后赋值给另一个变量
>>> print (c[::2])     # 打印偶数下标的元素
[9, 7, 5, 3, 1]
```

以上使用了 “`b = a[:]`” 的赋值形式。如果用 “`b = a`” 直接赋值，则列表 a 和 b 是同一个对象，改变其中任何一个的元素，另一个也会随之发生变化。

3.1.3 列表的运算

列表的运算主要包括相加、倍乘、片段、元素增减等等。使用基本符号运算时，列表只定义了加法运算 “+” 和倍乘运算 “*”。加法运算中，列表只能与列表相加，非列表的对象必须转换成列表之后才能与列表进行加法运算；列表乘以一个正整数 (倍乘) 等效于多个列表累加。将一个元素添加到列表的首部或尾部，可以先将该元素转换成列表再进行加法运算。如果添加在其他位置，则必须通过函数实现。

表7是 Python 内建的列表函数。

[1] 按正常的语言习惯，倒数第一个指的是最后一个。

表 7: 与列表操作相关的函数

函数	功能	说明
list(object)	将一个可迭代的对象转换成列表	L=list() 创建一个空列表
append(object)	在列表最后添加一个元素	
clear()	清空列表	
copy()	硬拷贝列表	如果直接使用 “=” 赋值，不会创建新对象
count(value)	返回列表中 value 的数目	
extend(object)	将一个迭代数的对象追加到列表中	
index(value, [start,[stop]])	返回第一个 value 在列表中的位置	
insert(index, object)	在 index 位置前插入一个元素	
pop(index)	返回 index 位置的元素，并将该元素从列表中删除	index 缺省时，表示最后一个元素
remove(value)	删除元素 value	
reverse()	列表反序	
sort(key, reverse)	根据 key 的规则排序，reverse 为 True 则反序	

可迭代对象是由多个元素组成的数据结构，包括字符串、序列、列表、元组、集合、词典等。字符串转换成列表时，其中的每一个字符成为列表中的一个元素。

表7中除了 list() 以外，其余都是列表对象的成员函数，意即 “`L=list([1,2,3])`” 创建一个列表 L，而 “`L.clear()`” 清除列表 L 中的元素。

下面的程序实现一组列表的操作：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

season = ['立春', '立夏', '立秋', '立冬']     # 创建一个列表
season.insert(1, '雨水')                      # 插入一个元素
season.extend(('小雪', '大雪'))               # 将元组元素追加到列表
index = season.index('立秋')                  # 得到``立秋''的下标
cell = season.pop(index)                      # 删除 index 处的元素
season[-1] = '大寒'                           # 修改最后一个元素
```

```
season.sort()
print ('清明' in season)
```

函数 copy() 可以创建一个新的列表对象，但如果列表中的元素是可迭代的数据结构时，这个结构在复制的列表和原列表中仍然是同一对象。看下面的例子：

```
lst1 = [['中秋', '重阳'], 8.15, 9.9]
lst2 = lst1.copy()
lst2[0][1] = '清明'
lst2[1] = 4.5
print (lst1)
```

完整地复制一个对象可以使用 copy 模块中的 deepcopy() 函数。

3.2 元组

元组 (tuple) 在很多性质上与列表完全一样，它也是一组有序数据的集合。唯一不同的是，它是一种不可变的数据结构，这意味着一旦赋值，该变量的成员不能单独修改，除非对这个变量重新赋值。

定义元组时，元组的元素之间用逗号分隔，如果只有一个元素，该元素后面也需要保留逗号，以区分表示元素本身的对象。有时候为了避免歧义，也可以在整组元素的外面加上括号 “()”。访问元组中的元素时，下标的使用方法与列表完全一致。

由于元组的不可变更属性，它的函数比列表的函数要少得多，index() 是一个较常用的元组函数，它用于获得元素所在的位置。

多数情况下，元组的功能都可以用列表来实现。少数情况 (如词典的关键字) 是列表不能替代的。此外由于元组是不可变数据，不用担心误操作更改了其中的元素。

3.3 集合

集合 (set) 包含一组无序的对象。与列表和元组不同，由于是无序数据，因此没有下标。与数学上的集合概念一样，集合中没有重复的元素。

集合使用 “{}” 符号包装元素[1]，元素之间用逗号分隔。集合的运算主要包括并集、交集、差异、查询等操作，表8列出了内建的集合函数。

表 8: 与集合操作相关的函数

函数	功能
set(object)	将一个可迭代的对象转换成集合
add(object)	向集合中添加一个元素
clear()	清空集合中的所有元素
copy()	硬拷贝集合。直接使用 “=” 赋值不会创建新对象

[1]注意，默认使用 “{}” 创建的不是空集合，而是空词典。

表 8: 与集合操作相关的函数 (续)

函数	功能
difference()	返回两个或多个集合的差集
difference_update()	按差集更新集合集
discard(value)	删除集合中的一个元素
intersection()	返回集合的交集
intersection_update()	按交集更新集合
isdisjoint()	判断两个集合是否有交集 (无交集返回 True)
issubset()	判断集合是否为另一个集合的子集
issuperset()	判断集合是否为另一个集合的超集
pop()	返回集合中的任意一个元素并将其从集合中删除
remove(value)	从集合中删除元素 value
symmetric_difference()	返回集合的对称差集
symmetric_difference_update()	以对称差集更新集合集
union()	返回若干个集合的并集
update()	更新到若干个集合的并集

表中除了函数 set() 以外，其它均为集合对象的成员函数。S=set() 创建一个空集合。

3.4 词典

词典 (dictionary)是通过关键字与对应值建立的一组数据的集合。关键字与值的对应关系用 ":" 表示，其中关键字必须是不可变数据类型，如数值、字符串、元组。每个元素之间用逗号分隔，整体则放在 "{ }" 中。

词典也是无序数据，访问词典中的元素不通过有序下标，而是通过关键字。词典中的关键字必须是唯一的。下面是一些词典的典型操作：

```
zhang3 = {}                    # 创建一个空词典，等效于 dict()
zhang3['name'] = 'Zhang3'      # 增加一个元素，关键字 ``name'',
                               # 值为 ``zhang3''
li4 = zhang3.copy()            # 复制到一个新的词典变量
li4['age'] = 23                # 增加一个新的元素
print (zhang3, li4)
```

同列表和集合类似，直接将词典使用 "=" 赋值不会创建新的对象。表 9 是词典对象的函数表。

表 9: 词典操作相关的函数

函数	功能	说明
dict(object)	将一个映射对象转换成词典	L=dict() 等效于 L={ }
clear()	清空词典	
fromkeys(keylist)	根据给出的关键字创建新词典	缺省方式下，值为 None
get(key[,d])	获得词典中关键字的值	关键字 key 不存在时返回 d
items()	返回词典 (key,value) 对应关系的列表	
keys()	返回词典中关键字的列表	
pop(key[, d])	删除关键字 key 对应项，返回对应值	删除关键字项还可使用内建关键字 “del”，但无返回值
popitem(key, value)	删除 (key, value) 对，将该项作为元组返回	
setdefault(key[,d])	添加/修改关键字 key 的值	
update(dict)	根据 dict 更新词典	
values()	返回词典中值的列表	

Python 2 曾使用 has_key(key) 函数来判断词典中是否存在关键字 “key” 项，现已逐渐用运算关系 “in” 取代，而 Python 3 则直接取消了这个函数。

```
zhang3 = dict(name='张三', age=23)
# 等效于 zhang3={'name': '张三', 'age': 23}
zhang3['phone'] = '010-87654321'
print (zhang3['gender'])
# ``gender'' 不存在，将触发异常错误
print (zhang3.get('gender', 'M'))
# 函数 get() 不会触发错误，缺省方式下返回 None
extra = {'gender': 'M'}
zhang3 = zhang3.update(extra)
# 将另一个词典并入
```

3.5 字符串对象

字符串 (str)在信息处理中是最基本也是最常用的对象，Python 为其构建了一整套处理函数。表10 – 12 列出了字符串对象的主要方法。

表 10: 字符串对象的常用函数–字符转换

函数	功能
capitalize()	将首字母大写，其它字母小写
lower()	将全部字母变成小写
upper()	将全部字母变成大写
swapcase()	将全部字母大小写互换
center(width[, fill])	使字符串居中，两边用 “fill” 填充
encode(encoding)	按 “encoding” 设定的方式编码
expandtabs(tabsize)	将字符串中的制表符转换成 “tabsize” 个空格
replace(old, new[, count])	字符串替换
title()	标题化字符串 (每个单词首字母大写)
ljust(width[,fill])	字符串左对齐，右边填充 “fill”
rjust(width[,fill])	字符串右对齐，左边填充 “fill”
lstrip(chars)	去掉字符串左边的 “chars” (缺省时去掉空格)
rstrip(chars)	去掉字符串右边的 “chars” (缺省时去掉空格)
strip(chars)	去掉字符串两边的 “chars” (缺省时去掉空格)
translate(table)	返回字符串在映射表 “table” 中的拷贝
zfill(width)	在数字字符串前面用 “0” 填充到长度 “width”

由于字符串变量的不可改变属性，字符转换的结果返回一个新的字符串，而原字符串变量不变。例如：

```
>>> a = 'hello, python'
>>> b = a.title()
>>> print (a + '\n' + b)
hello, python
Hello, Python
```

表 11: 字符串对象的常用函数–字符特征

函数	功能
isalnum()	是否都是是字母或数字

表 11: 字符串对象的常用函数–字符特性 (续)

函数	功能
isalpha()	是否都是字母
isdigit()	是否都是数字 (包括编码数字，罗马数字)
islower()	是否都是小写
isspace()	是否都是空格
istitle()	是否符合标题格式
isupper()	是否都是大写
isidentifier()	是否为识别符 (语法关键字)
isdecimal()	是否是数字 (不包括包括罗马数字)
isnumeric()	是否是数字 (不包括字节数字)
isprintable()	是否都是可打印字符
endswith(suffix)	是否以 “suffix” 结尾
startswith(prefix)	是否以 “prefix” 开头

表 12: 字符串对象的常用函数–查找与处理

函数	功能
count(sub[,start, end])	返回子串 “sub” 的数目
join(iterable)	将可迭代数据 “iterable” 用字符串串接
partition(sep)	以 “sep” 为中心将字符串分割成首尾三段
rpartition(sep)	类似 partition，但是从字符串尾部开始
split(sep, max)	以 “sep” (或空字符) 为分隔符将字符串分割成列表
rsplit(sep, max)	与 split() 类似，但方向是从后往前
splitlines()	以断行符为特征将字符串分割成列表
find(sub[,start, end])	查找子串 “sub” 的位置，不存在时返回 -1
rfind(sub[, start, end])	与 find() 功能类似，但是从字符串尾部开始
index(sub[,start, end])	与 find() 功能类似，子串 “sub” 不存在时报错
rindex(sub[, start, end])	与 index() 类似，从字符串尾部开始
format(...)	将变量进行格式化输出

format() 常用于格式化显示输出。下面是一个例子：

```
>>> "{0} + {1} = {2}".format(3, 4, 3 + 4)
'3 + 4 = 7'
```

“{ }” 中的数字对应 format() 参数序号，缺省时从 0 开始自然递增。

3.6 对象

Python 内建了很多结构化数据，面对不同的场合使用提供了很大方便。Python 是面向对象的程序，所有数据结构都是对象，包括在数值计算中使用的整数、浮点数也是对象。表 13 归纳了 Python 常用内建对象及其创建函数。

表 13: 常用内建对象

对象名	列表	元组	集合	词典	字符串	整数	浮点数
创建函数	list()	tuple()	set()	dict()	str()	int()	float()

函数 isinstance() 用于判断某个对象是否属于某个类的实例，类名称与创建函数的字母所对应的名称一致：

```
>>> a = '123'
>>> isinstance(a, int)
False
>>> isinstance(float(a), float)
True
```

4 程序结构

4.1 顺序结构

我们先从一个简单的程序开始，理解 Python 程序的书写方法：

清单 3: prime.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  def is_prime(number):
5      '''
6      returns True if number is a prime, False otherwise
7      >>> is_prime(5)
8      True
9      >>> is_prime(35)
10     False
11     >>> is_prime(203)
12     True
```

```
13      >>> [x for x in range(900, 1000) if is_prime(x)]
14      [907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]
15      '''
16      print (a)
17      for i in range(2, number):
18          if number % i == 0:
19              return False
20      return True
21
22  if __name__ == '__main__':
23      a = 127
24      if is_prime(a):
25          print("%d is a prime" % a)
26      else:
27          print("%d is not a prime" % a)
```

Python 根据源程序的书写顺序执行，没有特殊的起始语句或起始函数。由于它是解释性语言，没有预处理过程，因此每条语句执行时，其中的所有符号必须是在之前已经声明或定义过的。这里所谓的 “前”、“后”，不一定是体现在书写顺序上，而是由运行顺序决定的。例如上面的例子中，程序从第 22 行开始运行，第 23 行定义了变量 a，然后调用子程序 is_prime()，子程序的第 16 行打印 a，此时 a 作为全局变量已经存在，因此这段程序没有问题。但如果将 23 行对变量 a 的赋值写在调用子程序 is_prime() 之后 (同时注意调整第 24 行函数参数问题)，则程序运行到第 16 行时就会给出错误提示 “NameError: global name ’a’ is not defined”。

这个程序作为模块使用时 (自建模块方法见5.2.1节)，由于导入时不会运行从第 22 行开始的代码，因此第 16 行处的变量 a 仍可能是未定义的。

Python 一行超过一条语句时，语句之间用分号分割。为保证源代码的可读性，一般不建议一行写太长的语句。Python 使用行首空格表现代码块的层次结构，因此对行首的空格有严格要求，对表达式之间的空格没有限制。每一个层次块由上一行语句末的冒号引导，整块代码向内缩进，结束缩进时意味着块的结束。在上面的例子中，第 4 行定义了子程序名称，到第 22 行处缩进结束，第 5–21 行表示这段子程序的完整内容。子程序第 17 行引导了一个循环，第 18–19 行则是这个循环体，循环中的一个条件判断又构成了一个块。

同一结构层次中要求相同的缩进，不同结构中的缩进可以不同，例如将第 23–27 行整体向前或向后移动几格都是可以接受的。

使用缩进时，制表符和空格不能混用。多数 Python 开发人员不建议使用制表符作为缩进。

有时候，为了源代码的紧凑，一些简短的块可以不另起行，直接写在块引导行的后面，例如第 24–25 行合并为一行，26–27 两行也可以合并为一行。

4.2 条件

与条件判断相关的关键字有 if、else、elif。条件语句通常有下面三种形式：

- 单独条件

```
a = '1234'
if isinstance(a, int):
    print ('a is an integer')
```

- 互斥条件

```
a = 2018
if a > 0:
    print ('a is a positive')
else:
    print ('a is not a positive')
```

- 多重条件

```
if a > 0:
    print ('a is a positive')
elif a < 0:
    print ('a is a negative')
else:
    print ('a is zero')
```

需要注意的是，elif 和 else if 不完全等价，else 必须构成一个新的块，也就是说 else 后面必须要有一个冒号，不能紧跟 if。

条件语句的另一种用法是直接夹在其他语句中，不作为块：

```
str = 'a is a positive' if a > 0 else 'a is not a positive'
```

4.3 循环结构

与循环相关的关键字有 for...in、while、continue、break。continue 用于跳过循环体后面的语句，进入下一轮。break 则是结束循环。

- for 循环

```
days = [('Jan', 31), ('Feb', 28), ('Mar', 31), ('Apr', 30)]
for x, y in days:
    print('There are {} days in {}'.format(y, x))
```

- while 循环

```
# 10 秒倒计时
import time
c = 10
while True:
    print (c)
    c -= 1
    if c <= 0: break
    time.sleep(1)
print ('Fire!')
```

在某些情况下，循环可以和 else 搭配使用：

```
lists = [3, 9, 4, 2, 0, 5]
for x in lists:
    if x <= 0:
        print ('Find a non-positive number')
        break
else:
    print ('All numbers in list are positive')
```

注意，这里的 else 与 for 属同一层，与循环中的 if 不在同一层。

与条件语句类似，循环也可以嵌在其他语句中，起到简化结构的目的：

```
# 构造10以内的平方表
lists = [x*x for x in range(10)]
```

4.4 子程序

关键字 def 定义子程序的函数名，函数的形式参数列表写在括号里。与子程序有关的另一个关键字是 return，用于向主程序提供返回值。

主程序向子程序传递参数有下面几种形式：

- 参数顺序一一对应。这种形式比较容易理解和接受，也是其他很多编程普遍采用的形式：

```
def add(x, y):
    return x + y

print (add (3, 5))                # This will print "8"
print (add ('Hello', 'world'))    # This will print "Helloworld"
```

- 为参数设定一个关键字，同时参数具有了一个缺省值。此时参数首先根据关键字对应，在不提供参数关键字时再根据顺序对应：

```
def add(string='hello', number=1):
    return string*number

print (add ())           # print "hello"
print (add (number=2))   # print "hellohello"
```

- 形式参数任意，实际参数由调用的主程序决定。这种形式可以和前两种结合使用，所有任意形式参数表放在形式参数的最后，作为元组传递给子程序：

```
def add(title, *args):
    print (title, end='=')
    return sum(args)

print (add('total',1,2,3,4))
# This will print "total=10"
```

- 形式参数任意，实际参数以关键字传递给子程序，所有关键字和对应值组成一个词典传递给子程序：

```
def add(**args):
    return sum(args.values())

print (add(a1=2, x=3, b=4, z3=-5))
# This will print "4"
```

子程序向主程序的返回值数目原则上应与主程序接收的变量数相对应：

```
def algorithm(a, b):
    return a+b, a-b

x, y = algorithm(20, 5)
```

当使用单个变量接收多个返回值时，返回值以元组的形式向变量赋值。

5 模块

模块是一组具有重复使用价值的功能性代码。合理地使用模块，可以简化程序设计过程，提高程序的可维护性，控制程序规模，同时还有助于减少错误。

5.1 模块的导入

导入模块的方法有以下几种：

1. 直接导入模块名，如：

   ```
   import math
   ```

   这种方法书写最简单，但使用最麻烦。模块中的所有符号 (函数名、变量名、常量名等等) 前面都需要加上模块名。

2. 导入指定的符号：

   ```
   from math import exp, pi, sin
   ```

   以后在使用这些符号时不再用模块名为前缀，直接使用符号名本身，如“`print (sin(pi/6))`”。

3. 导入符号，并将其重新命名，如：

   ```
   from math import exp as EXP
   ```

   之后使用模块中的符号时只能用重命名的形式，如“`print (EXP(2.0)/10)`”。重命名符号可能由于个人习惯的需要，也可能是出于代码兼容性方面的考虑。

4. “`from module import symbol`”这种形式只能使用模块中已导入的符号，例如在上面的两种方法中，虽然我们知道 math 模块中还有 $\cos(x)$ ，但也不能直接使用，必须加入导入列表才能使用。一个比较粗暴的方式是：

   ```
   from math import *
   ```

   它导入除下划线“_”开头的所有符号。星号“*”在这里起到的是通配符的作用。不论该模块里有什么符号，都可以以原符号形式使用，不需要加前缀。虽然方便，但却是不太推荐的一种方式。它容易引起符号使用的混淆。

在 Python 环境中使用命令 help('modules') 可以列出当前环境下支持的所有模块。如果需要了解某个模块的具体用法，可以在导入后使用 help(mod_name) 查看，也可在命令行中使用 pydoc mod_name 查看。

5.2 创建自己的模块

5.2.1 以 py 文件作为模块

所有 Python 文件都可以作为模块使用，其中的符号可以通过 import 被其他模块导入。为避免导入时运行不该运行的代码，可以利用一个特殊的变量“__name__”将主程序与可导入符号分离。例如：

清单 4: fibo.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print (b, end=', ')
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == '__main__':
    fib(30)
```

上面的程序可以独立运行：

```
$ python3 fibo.py
1, 1, 2, 3, 5, 8, 13, 21,
```

同时，另一个模块可以使用 fibo.py 中的函数：

清单 5: func.py

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from fibo import fib2

if __name__ == '__main__':
    print(fib2(200))
```

运行结果如下：

```
$ python3 func.py
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

同样，func.py 也可以继续作为模块，供其他模块使用。尽管在这个例子中，func.py 没有实质性内容，但它导入的模块 fib2 可以传递到上层。

当给定一个模块名后，Python 首先寻找是否存在该名称的内建模块，如果没有，则按 sys.path 列出的目录表顺序查找。通常，当前目录是这个表中的第一项，因此，置于当前目录的文件总是以第一顺序被作为模块导入。

5.2.2 以目录名作为模块

在目录下创建一个文件 "__init__.py"，哪怕就是一个空文件，该目录就成为一个 Python 模块名，其下的 py 文件和目录则成为该模块的子模块，这使得模块更加便于开发和维护。导入模块时将首先运行 __init__.py。__init__() 是对象的初始化函数，一些初始化工作可以写在这个函数里。__init__.py 承担的是同样的功能。

例如，我们可以这样构建一个模块目录：

```
fibo                (目录)
  |
  +-- __init__.py   (空文件)
  +-- fib.py        (函数fib())
  +-- fib2.py       (函数fib2())
```

当目录 fibo 属于 sys.path 的一部分时，可以通过下面的方式使用：

```
from fibo import fib, fib2
a = fib.fib(30)
b = fib2.fib2(100)
```

6 编写图形界面程序

6.1 Tkinter

Python 的出发点不是图形化，因此没有内建的图形用户接口函数。但图形化又是如此重要，以至于很多开发人员为 Python 提供了不同的 GUI (Graphical User Interface，图形用户接口) 工具包，其中比较著名的有 PyGTK、PyQT、Tkinter 等等。

其中 Tkinter 在编译 Python 的时候就可以获得，很多 Python 发布版默认就已经包含了这个模块 (该模块在 Python 3 中名称是 tkinter，Python 2 中的名称是 Tkinter)，无需额外安装其他软件包，依赖关系简单，使用较方便。

6.2 Tkinter 的使用

一个使用 Tkinter 的 (最小) 图形化界面如下:

表 14: 几种 Python 图形包

模块	说明	特点
Tkinter	基于 Tcl/Tk 库	Python 的半标准化图形库
wxPython	基于 wxWindows 系统	跨平台
PythonWin	使用 Windows GUI	仅用于 Windows 操作系统
PyGTK	基于 GTK 库	主要见于 Linux 系统
PyQt	基于 Qt 库	跨平台

清单 6: button.py

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3
4  import tkinter
5
6  def func():
7      print('Button Pressed')
8
9  win = tkinter.Tk()
10 win.geometry('200x100+300+400')
11 win.title('tkinter')
12 hello = tkinter.Button(win,
13                         text='Hello',
14                         command=func)
15 hello.pack(expand=True, padx=20, pady=10)
16 win.mainloop()
```

下面是逐行解释：

- 第 4 行，导入 Tkinter 模块；
- 第 6–7 行，定义一个函数，供下面调用；
- 第 9 行，创建顶层窗口；
- 第 10 行，设置窗口大小和位置。该项设置在多数情况下不是必须的，窗口会根据内部的组件自行调整大小；
- 第 11 行，设置窗口标题；
- 第 12–14 行，创建一个按钮组件，其容器是 win 对象，按钮上的文字使用参数 “text” 设置，按钮动作 (响应函数) 通过 “command” 设置；

- 第 15 行，将组件 hello 加入容器，并设置与容器的衔接关系。
- 第 16 行，所有顶层窗口显示，进入消息循环，此后各组件并行工作，接收消息，做出反应。由于在这个例子中只创建了一个顶层窗口 win，因此只显示这一个窗口。

上面的例子运行时，将在屏幕坐标 (300, 400) 处显示 200×100 大小的窗口，窗口内有一个按钮，鼠标点击按钮时会在终端打印一串文本。

由此可以归纳 Tkinter 图形化编程步骤 (使用其他图形化模块的步骤大同小异)：

1. 导入 Tkinter 模块；
2. 创建顶层窗口；
3. 创建 GUI 组件，设计其样式、响应函数；
4. 将组件组装进窗口；
5. 调用 mainloop()，进入主循环。

同其他 GUI 程序一样，GUI 库只负责窗口内的组件，窗口样式则由窗口管理器负责。换句话说，在不同的系统中，同一个程序的标题栏、边框、窗口控制按钮等表现形式可以是不同的。

6.3 常用组件

表15列出了比较常用的 Tkinter 组件。

表 15: Tkinter 常用组件

组件名称	说明
Label	标签文字或图像，一般不响应消息
Message	消息框，与 Label 功能类似，用于多行文本显示
Button	按钮，与标签类似，但通常需要响应鼠标及键盘消息事件
Checkbutton	复选按钮，使用两个不同状态反映一项功能有/无
Radiobutton	单选按钮，由一组互斥的按钮组成，其中有且只有一个被选中
Entry	单行文本输入框，提供一个可编辑的文本输入环境
Text	多行文本输入框
Scale	标尺，通过滑块设置/获取数值，有水平方向和垂直方向两种
Scrollbar	滚动条
Canvas	画布，提供画图环境
Frame	框架，本身不可见，通常作为其他组件的容器
Menu	菜单

下面是一个组合了画布、标尺、按钮、文本框的示例程序。

清单 7: canvas.py

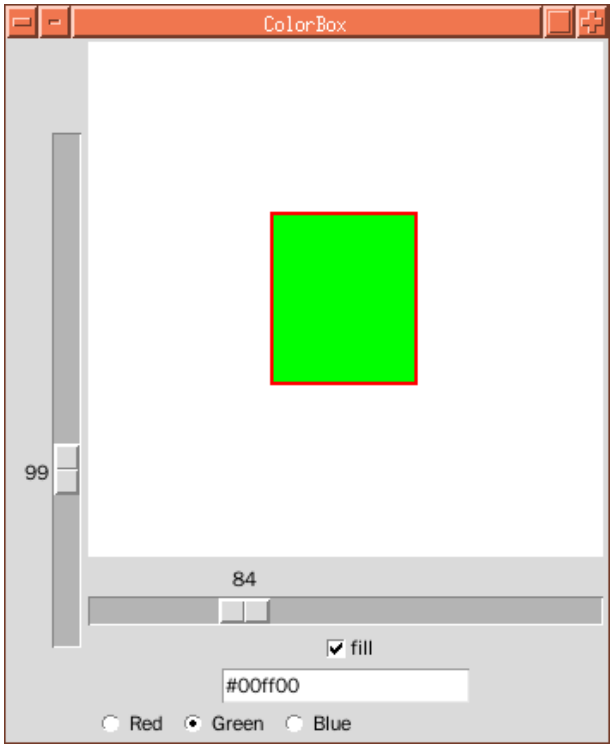
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from tkinter import *

# change height of the rectangle
def resizev(v):
    height = int(v)
    width = scaleh.get()
    canvas.coords(box, 150 - width/2,
                       150 - height/2,
                       150 + width/2,
                       150 + height/2)
    if filled.get():
        canvas.itemconfig(box, fill=rgbv.get())
    else:
        canvas.itemconfig(box, fill='white')

# change width of the rectangle
def resizeh(v):
    width = int(v)
    height = scalev.get()
    canvas.coords(box, 150 - width/2,
                       150 - height/2,
                       150 + width/2,
                       150 + height/2)
    if filled.get():
        canvas.itemconfig(box, fill=rgbv.get())
    else:
        canvas.itemconfig(box, fill='white')

# set color by RGB
def setcolor():
    if color.get() == 0:
```

```
35          rgbv.set('#ff0000')
36      elif color.get() == 1:
37          rgbv.set('#00ff00')
38      elif color.get() == 2:
39          rgbv.set('#0000ff')
40
41  win = Tk()
42  win.title('ColorBox')
43
44  # create 2 frames as layout
45  frame1 = Frame(win).pack(side=RIGHT)
46  frame2 = Frame(win).pack()
47
48  # create a rectangle
49  canvas = Canvas(frame1, width=300, height=300,bg='white')
50  box = canvas.create_rectangle(150, 150, 150, 150,
51                                width=2,
52                                outline='red',
53                                fill='white')
54
55  # horizontal scale
56  scaleh = Scale(frame1, length=300,
57                         from_=0,
58                         to=300,
59                         orient=HORIZONTAL,
60                         command=resizeh)
61  # vertical scale
62  scalev = Scale(frame1, length=300,
63                         from_=300,
64                         to=0,
65                         orient=VERTICAL,
66                         command=resizev)
67
68  filled = IntVar()
69  # set if the box is filled with color or not
70  fill = Checkbutton(frame2, text='fill',
71                             onvalue=1,
72                             offvalue=0,
```

```
73                              variable=filled)
74
75  # set fill color in RGB
76  color = IntVar()
77  color.set(0)
78  red   = Radiobutton(frame2, text='Red',
79                              variable=color,
80                              value=0,
81                              command=setcolor)
82  green = Radiobutton(frame2, text='Green',
83                              variable=color,
84                              value=1,
85                              command=setcolor)
86  blue  = Radiobutton(frame2, text='Blue',
87                              variable=color,
88                              value=2,
89                              command=setcolor)
90
91  rgbv = StringVar()
92  rgbv.set('#ff0000')
93  rgb = Entry(frame2, textvariable=rgbv)
94
95  scalev.pack(side=LEFT)
96  canvas.pack()
97  scaleh.pack()
98  fill.pack(side=TOP)
99  rgb.pack(side=TOP)
100
101 red.pack(side=LEFT)
102 green.pack(side=LEFT)
103 blue.pack(side=LEFT)
104
105 win.mainloop()
```

程序界面如图4。

图 4: Tkinter 组件的组合使用



7 树莓派外设

各种设备的驱动程序属于 Linux 内核的一部分。设备驱动被加载后，会在 /dev 目录下创建一个设备文件，用户程序通过读写这个设备文件实现对设备的操作。在树莓派内核中，提供了一些较常用设备的用户空间操作方法。

7.1 GPIO

GPIO (General Purpose Input/Output，通用 I/O 接口)通过一只引脚实现简单的输入/输出功能。用户空间可以通过下面简单的方法使用 GPIO。

首先，在 /sys/class/gpio 目录中，可以看到有这样几个文件和目录：

```
/sys/class/gpio       (目录)
  |
  +-- export          (用于创建GPIO节点)
  +-- unexport        (用于删除GPIO节点)
  +-- gpiochip0       (符号链接)
  +-- gpiochip100     (符号链接)
  +-- gpiochip128     (符号链接)
```

我们重点关心 export 和 unexport 这两个文件。

文件 export 用于创建一个 GPIO 的目录 (节点)，该目录提供了对应引脚输入/输出功能的进一步操作方法。为达到这个目的，首先确定我们想使用的 GPIO 序号。这里的序号指的是芯片内部的 GPIO 编号，不是电路板插座上的编号。例如我们想使用 GPIO18，它对应树莓派 B+ 的 40 脚排针的第 12 脚位置。创建 GPIO18 的功能可以通过向文件 export 写入一个数字 “18” 实现：

```
# echo 18 > export
```

命令执行后，会在该目录下看到多出一个链接文件 gpio18 (实际上是一个目录)。进入这个目录，可以看到有 value 和 direction 等文件。direction 用于设置 GPIO 的输入或输出方式，它接受两个值：“in” 和 “out”；value 用于获取或控制对应引脚的电平，是一个可读文件，如果里面的值是 0，表示当前引脚电平为低，如果是 1 表示电平为高。

有了上面的准备，我们就可以实现一些简单的控制功能了。下面是一个实现交替输出高低电平的 Python 程序：

清单 8: gpio.py

```
#!/usr/bin/python3
# -*- encoding: utf-8 -*-

import time

f = open('/sys/class/gpio/export')
f.write('18')
f.close()
f = open('/sys/class/gpio/gpio18/direction')
f.write('out')
f.close()

f = open('/sys/class/gpio/gpio18/value')
while True:
    f.write('0\n')
    time.sleep(0.5)
    f.write('1\n')
    time.sleep(0.5)
```

在 GPIO18 对应的排针第 12 引脚接上 LED 发光二极管，运行上面的程序，可以看到发光管以每秒一次的节奏闪烁。

GPIO 不再使用时，应向文件 unexport 写入对应的序号将节点删除。已经存在的节点不能重复创建，否则会产生文件操作错误。

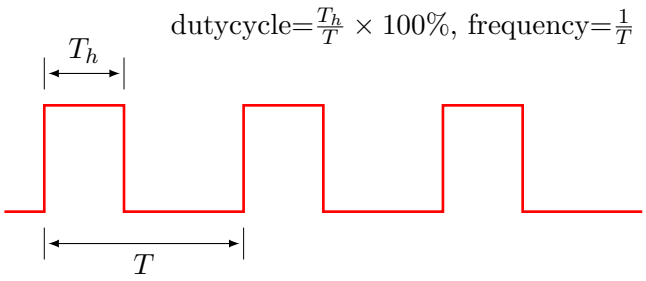
嵌入式处理器通常都有大量的 GPIO 引脚，有些引脚是功能复用的，使用前需要先明确该引脚未用于其他功能，否则会造成系统故障。

7.2 PWM

PWM (Pulse Width Modulation，脉冲宽度调制)，或称脉冲时间调制 (Pulse Duration Modulation, PDM)，是一种使用脉冲信号进行信息编码的技术。[1]. 它可以产生可变占空比的矩形波，通过数字方法实现模拟控制信号。

在上面的 LED 控制程序中，改变两个 sleep() 函数的时间参数，并将一个周期的总时间缩短到 0.1 秒以下，此时肉眼无法分辨 LED 的闪烁，看到的就是 LED 不同的亮度。这实际上就是软件实现的 PWM 形式。

软件实现的 PWM，频率不能太高，处理器负担加重，精度也会有较大的误差。



要启用硬件，需要将设备树 pwm-2chan 加入系统启动设置文件 config.txt 中，该文件位于引导分区：

```
dtoverlay=pwm-2chan,pin=18,func=2,pin2=19,func2=2
```

并在内核中使能 PWM-bcm2835 (可以是模块加载，也可以直接由内核支持)。此设置同时也会将 GPIO18 和 GPIO19 连向耳机插座。因此如果 PWM 输出的是音频信号时，插上耳机就可以听到。

硬件 PWM 使用硬件定时器产生高频率准确周期的脉冲信号，精确控制引脚输出高低电平的时间。树莓派有两个硬件 PWM 模块，可以通过配置引导参数激活其功能。当硬件 PWM 功能可用时，可以在 /sys/class/pwm 目录下看到 pwmchip0 的链接 (目录)。这个目录里有下面这些文件：

```
/sys/class/pwm/pwmchip0      (目录)
  |
  +-- export                 (用于创建 PWM 节点)
  +-- unexport               (用于删除 PWM 节点)
  +-- npwm                   (PWM 数目)
  +-- uevent
```

npwm 显示的是 2，因此可以通过 “echo 0 > export” 和 “echo 1 > export” 创建两个 PWM 节点，它们分别对应 GPIO18 和 GPIO19 (也可能对应 GPIO12 和 GPIO13，取决于系统启动时向内核传递的参数设置)。此时引脚的 GPIO 功能便切换到 PWM 功能。

在 pwm0 或 pwm1 目录下有这几个重要的文件：

[1] wikipedia

```
/sys/class/pwm/pwm0       (目录)
  |
  +-- enable              (使能/禁止 PWM 输出)
  +-- capture             (PWM 捕获)
  +-- period              (周期设置，单位 ns)
  +-- duty_cycle          (占空周期设置，单位 ns)
  +-- polarity            (极性设置，normal/inversed)
```

占空周期和周期的参数以 ns 为单位，duty_cycle/period 即为占空比 。设置完这两个参数后，向 enable 写入 1 即启动 PWM 输出。polarity 用于控制输出的极性，接受两个控制值：normal 和 inversed。正常情况下 (normal) duty_cycle 是高电平的持续周期，当写入 inversed 后输出波形颠倒。

下面的操作将产生 500Hz、50% 占空比的矩形波：

```
# echo "2000000" > period
# echo "1000000" > duty_cycle
# echo 1 > enable
```

PWM 模块不再使用时，应向 unexport 写入 PWM 通道号 (0 或 1)，以删除 pwmX 节点。

8 树莓派 I/O 模块

8.1 GPIO pins

注意不同版本的树莓派I/O 模块引脚的差别。表16是树莓派 2 和树莓派 3 的引脚功能。

8.2 Python GPIO 模块安装

为树莓派 GPIOGPIO 开发的 Python 模块名为 RPi.GPIO。

8.2.1 从软件仓库安装

如果树莓派的操作系统是 RaspbianRaspbian，缺省方式下 RPi.GPIO 已经安装了。为了确保安装的是最新版本，不妨尝试进行如下操作：

```
$ sudo apt-get update
$ sudo apt-get install python-rpi.gpio python3-rpi.gpio
```

以上安装包含了 Python 2 和 python 3 两个版本的 GPIO 模块。

树莓派的其他发行版，可以用 Python 安装包管理工具 pip 安装 GPIO 模块：

```
# pip install RPi.GPIO
```

[1]http://wiringpi.com

表 16: WiringPi: GPIO Pin Numbering[1]

P1: The Main GPIO connector							
WiringPi	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi
		3.3v	1	2	5v		
8	GPIO2	SDA1	3	4	5v		
9	GPIO3	SCL1	5	6	GND		
7	GPIO4	GCLK	7	8	TxD	GPIO14	15
		GND	9	10	RxD	GPIO15	16
0	GPIO17	GEN0	11	12	GEN1	GPIO18	1
2	GPIO27	GEN2	13	14	GND		
3	GPIO22	GEN3	15	16	GEN4	GPIO23	4
		3.3v	17	18	GEN5	GPIO24	5
12	GPIO10	MOSI	19	20	GND		
13	GPIO9	MISO	21	22	GEN6	GPIO25	6
14	GPIO11	SCLK	23	24	CE0	GPIO8	10
		GND	25	26	CE1	GPIO7	11
	ID-SD		27	28		ID-SC	
	GPIO5		29	30	GND		
	GPIO6		31	32		GPIO12	
	GPIO13		33	34	GND		
	GPIO19		35	36		GPIO16	
	GPIO26		37	38		GPIO20	
		GND	39	40		GPIO21	
WiringPi	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi

8.2.2 从源码开始安装

首先从 sourceforge 网站下载源代码压缩包：[2]

```
$ wget https://sourceforge.net/projects/raspberry-gpio-python/files/\
latest/download -O RPi.GPIO-0.6.3.tar.gz
```

然后在本地解压、编译、安装。以下是安装步骤：

```
$ tar xf RPi.GPIO-0.6.3.tar.gz
$ cd RPi.GPIO-0.6.3
$ python3 setup.py build
```

[2]反斜线 “\” 在命令行操作中表示换行。Linux 命令行环境中，一行命令可容纳有足够长的空间，通常情况下不必强制换行。此处换行仅出于排版要求。

```
# python3 setup.py install
```

以上步骤完成 Python 3 的 GPIO 模块。如果希望将 GPIO 模块用于 Python 2，应将上面的 python3 换成 python 或 python2 完成操作。由于源码是用 C 语言写成的，因此本地编译过程中要求树莓派已经安装了 GCC 编译工具。安装时需要将一些文件复制到系统目录，因此上述过程最后一步 (安装) 要求超级用户权限。

8.3 GPIO 模块基本使用放法

8.3.1 GPIO 初始化

与使用其他模块一样，Python 首先应导入 GPIO 模块。为了简化模块书写，也是为了保持在不同平台上的兼容性，习惯上将 RPi.GPIO 重命名为 GPIO：

```
import RPi.GPIO as GPIO
```

下面的代码可以检查模块导入是否成功：

```
try:
    import RPi.GPIO as GPIO
except RuntimeError:
    print('''
        Error importing RPi.GPIO!  This is probably
        because you need superuser privileges.  You
        can achieve this by using 'sudo' to run your
        script''')
```

树莓派上，RPi.GPIO 提供了两种标记 I/O 引脚的方式：

1. BOARD 模式：按板卡插脚编号，即 1 号脚是第 1 排左边、2 号脚是第 1 排右边、...、40 号脚是第 20 排右边。这种标记方式，在使用 GPIO 模式设置函数 setmode() 时，通过输入参数 `BOARD` 指定。BOARD 模式的优点是，不论使用哪个版本的树莓派，GPIO 的编号是一样的，保持了源代码的兼容。

2. BCM(BroadCom Mode) 模式：按博通 SOC 芯片的 GPIO 序号编号。这种方式，必须根据电路设计图找到插脚和芯片引脚的对应关系 (图16)。在某一个版本的树莓派上正常运行的程序，换到另一个版本的树莓派上可能会出问题。

使用 GPIO 功能之前，应用下面的函数设置引脚模式：

```
GPIO.setmode(GPIO.BOARD)
# or
GPIO.setmode(GPIO.BCM)
```

要检查的当前使用的是哪一种引脚编号模式，可使用下面的函数：

```
mode = GPIO.getmode()
```

返回值 mode 可能是 GPIO.BOARD、GPIO.BCM 或者 None。

系统有可能同时运行多个程序。如果 RPi.GPIO 模块检测到与当前设置模式冲突，会发出警告。有些警告可能无伤大雅。避免发出警告的的方法是用下面的方法： `GPIO.setwarnings(False)` 建议不要忽略任何警告。

8.3.2 引脚功能设置

每只引脚使用前都应明确其 I/O 功能：作为输入还是作为输出。

```
GPIO.setup(channel, GPIO.IN)
# or
GPIO.setup(channel, GPIO.OUT)
```

参数 channel 是 GPIO 的通道编号，依所设定的引脚编号模式不同而定 (BOARD 模式或者 BCM 模式)。下文对此不再重复说明。

更多有关输入通道的内容在 8.4 节将详细介绍。

当某通道用于输出功能时，还可以在功能设置的同时为其设定输出值：

```
GPIO.setup(channel, GPIO.OUT, initial=GPIO.HIGH)
```

参数 initial 的取值有 GPIO.LOW 和 GPIO.HIGH，分别对应输出低电压和高电压。

函数 setup支持多个通道同时设置：

```
chan_list = [11, 12]    # add as many channels as you want!
                        # you can tuples instead i.e.:
                        #    chan_list = (11, 12)
GPIO.setup(chan_list, GPIO.OUT)
```

8.3.3 输入方式

函数input() 从输入通道读取当前引脚状态：

```
state = GPIO.input(channel)
```

返回值state 可能是 0 或 1 (对应GPIO.LOW 或GPIO.HIGI)，取决于引脚实际逻辑电平状态。

8.3.4 输出方式

将某通道channel 输出逻辑电平，通过函数output() 实现：

```
GPIO.output(channel, state)
```

参数state 可以是 0/GPIO.LOW/False 或者1/GPIO.HIGH/True。

8.3.5 多通道输出

也可以一次调用实现多个通道的输出控制：

```
chan_list = [11, 12]
# also works with tuples
GPIO.output(chan_list, GPIO.LOW)
# sets all to GPIO.LOW
GPIO.output(chan_list, (GPIO.HIGH, GPIO.LOW))
# sets first HIGH and second LOW
```

8.3.6 模式清除

程序结束时，应清除所有用到的资源，将其复原。复原后，所有 GPIO 通道处于输入方式，无上拉/下拉，避免无意中把 GPIO 引脚短路造成损坏：

```
GPIO.cleanup()
```

cleanup()仅清除程序中用到的 GPIO 通道，它同时也清除引号编号模式。

如果程序结束时不打算清除所有通道，也可以清除部分通道的功能，包括清除单个通道、多个通道的 list 或 tuple 结构：

```
GPIO.cleanup(channel)
GPIO.cleanup( (channel1, channel2) )
GPIO.cleanup( [channel1, channel2] )
```

8.3.7 其他信息

树莓派信息：

```
GPIO.RPI_INFO
```

树莓派版本信息：

```
GPIO.RPI_INFO['P1_REVISION']
GPIO.RPI_REVISION    (deprecated)
```

RPi.GPIO 模块版本信息:

```
GPIO.VERSION
```

8.4 输入

检查 GPIO 输入状态的方式有好几种。第一种、也是最简单的一种方法是即时检查引脚的输入值，也就是查询方式。当读取状态的指令没有赶在正确的时间上时，查询方式有可能错过状态的变化。查询方式常常用在循环中，因此也会增加处理器的负担。另一种就是使用中断方式 (边沿检测)。边沿是指电平从高到低 (下降沿) 或从低到高 (上升沿) 变化的瞬间。

8.4.1 上拉/下拉电阻

如果输入端不连接任何元件，它被称为是浮空的。换句话说，读取这个输入端引脚的电平值是不确定的，因为它没有受到明确的电压控制。在受到干扰时输入值可能会发生变化。

为避免这一现象的发生，通常会在输入端连接一个上拉或下拉电阻。这时，在未受到其他控制信号的作用时的电压是确定的。硬件设计的上拉/下拉电阻可通过软件控制。硬件上，将一个 10kΩ 的电阻置于输入通道和 3.3V 电源之间 (上拉)，或置于输入通道和地之间 (下拉)。GPIO 模块通过软件控制上拉或下拉状态：

```
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
# or
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

8.4.2 查询方式测试

下面的代码用来抓取某一瞬间的是输入状态：

```
if GPIO.input(channel):
    print('Input was HIGH')
else:
    print('Input was LOW')
```

在循环中使用查询方式 (假设捕获从低电平到高电平变化的状态)：

```
while GPIO.input(channel) == GPIO.LOW:
    # wait 10 ms to give CPU chance to do other things
    time.sleep(0.01)
```

8.4.3 中断方式

通常我们更关心输入通道的状态变化，而不是电平本身。这种状态变化，我们成之为事件。在程序忙于其他事务时，为了避免错过按键动作，这里提供两种解决方案：

1. `wait_for_edge()` 函数

   `wait_for_edge()` 函数设计用于程序阻塞，直到某个预期的边沿变化被检测到。上面那段循环检测按键的代码可以写成：

```
GPIO.wait_for_edge(channel, GPIO.RISING)
```

可被检测的状态有GPIO.RISING、GPIO.FALLING 或者GPIO.BOTH。这种方法的优点是，它占用的 CPU 资源非常少，CPU 可以省下时间做其他任务。

如果只是想等待一段确定的时间，可以利用参数timeout：

```
# wait for up to 5 seconds for a rising edge
# (timeout is in milliseconds)
channel = GPIO.wait_for_edge(channel,
                             GPIO_RISING,
                             timeout=5000)
if channel is None:
    print('Timeout occurred')
else:
    print('Edge detected on channel', channel)
```

2. event_detected() 函数。当事件被检测到时，安排一个回调函数处理该事件。

使用 event_detected()，处理器忙于其他事务时不会错过输入引脚状态的变化。这在使用 PyQt 或者 Tkinter 这类进入主循环监听事件的 GUI 程序里会非常有用。

```
GPIO.add_event_detect(channel, GPIO.RISING)
# add rising edge detection on a channel
do_something()
if GPIO.event_detected(channel):
    print('Button pressed')
```

可检测的事件有GPIO.RISING、GPIO.FALLING 或者GPIO.BOTH。

8.4.4 回调函数

RPi.GPIO 为回调函数安排一个线程，这意味着回调函数和主程序并行运行，可以即时响应边沿变化，例如：

```
def my_callback(channel):
    print('This is a edge event callback function!')
    print('Edge detected on channel %s'%channel)
    print('This is run in a different thread to your
           main program')

# add rising edge detection on a channel
GPIO.add_event_detect(channel,
```

```
                        GPIO.RISING,
                        callback=my_callback)
    #...the rest of your program...
```

多个回调函数的方法：

```
    def my_callback_one(channel):
        print('Callback one')

    def my_callback_two(channel):
        print('Callback two')

    GPIO.add_event_detect(channel, GPIO.RISING)
    GPIO.add_event_callback(channel, my_callback_one)
    GPIO.add_event_callback(channel, my_callback_two)
```

这种情况，回调函数不是同步执行，而是按加入的先后顺序执行。

8.4.5 按键去抖动

可能你会注意到，每次按键动作，回调函数会被多次调用，其原因是由按键抖动 导致的 (每次按键操作，按键会由于机械设计的原因产生若干次通断)。解决抖动问题有三种方法 (实际上是两种)：

- 在开关上接一个 0.1uF 的电容
- 延迟读按键状态，等待其稳定 (软件去抖动)
- 软、硬件结合

用软件方法去抖动，就是在指定回调函数时使用参数bouncetime，为它设一个延迟时间 (ms)，如：

```
# add rising edge detection on a channel, ignoring further
# edges for 200ms for switch bounce handling
GPIO.add_event_detect(channel,
                      GPIO.RISING,
                      callback=my_callback,
                      bouncetime=200)
# or
GPIO.add_event_callback(channel,
                        my_callback,
                        bouncetime=200)
```

8.4.6 删除事件检测

如果出于某种原因，程序不再需要边沿检测了，可通过下面的方法删除它：

```
GPIO.remove_event_detect(channel)
```

8.5 GPIO 输出

GPIO 输出工作方式按如下过程操作：

1. 首先，设置 RPi.GPIO 工作方式 (见8.3节).

   ```
   import RPi.GPIO as GPIO
   GPIO.setmode(GPIO.BOARD)
   GPIO.setup(12, GPIO.OUT)
   ```

2. 设置输出电平：

   ```
   GPIO.output(12, GPIO.HIGH)
   # or
   GPIO.output(12, 1)
   # or
   GPIO.output(12, True)
   # or
   GPIO.output(12, GPIO.LOW)
   # or
   GPIO.output(12, 0)
   # or
   GPIO.output(12, False)
   ```

3. 同时输出多个通道：

   ```
   chan_list = (11, 12)
   # all LOW
   GPIO.output(chan_list, GPIO.LOW)
   # first LOW, second HIGH
   GPIO.output(chan_list, (GPIO.HIGH, GPIO.LOW))
   ```

4. 程序结束时完成清除工作：

   ```
   GPIO.cleanup()
   ```

根据读取到的输入状态的状态设置输出状态，例如，输出状态切换：

```
GPIO.output(12, not GPIO.input(12))
```

8.6 使用 PWM

一个 PWM 波形包含两个参数: 频率 (以 Hz 为单位) 与占空比 (以百分比为单位)。

创建一个 PWM 实例:

```
p = GPIO.PWM(channel, frequency)
```

启动 PWM:

```
p.start(dc)
```

参数dc 是占空比 ($0.0 \leq dc \leq 100.0$)

改变频率：

```
p.ChangeFrequency(freq)
```

参数freq 是新的频率值 (单位 Hz).

改变占空比：

```
p.ChangeDutyCycle(dc)  # where 0.0 <= dc <= 100.0
```

停止 PWM:

```
p.stop()
```

如果变量p 的参数范围超出，PWM 也会停止。

下面是一个控制 LED 两秒钟闪一次的例子，假设 GPIO 通道 12 接有 LED 发光管：

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)

p = GPIO.PWM(12, 0.5)
p.start(1)
input('Press return to stop:')   # raw_input() in Python 2
p.stop()
GPIO.cleanup()
```

下面是一个 LED 明暗渐变的例子:

```
import time
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(12, GPIO.OUT)

p = GPIO.PWM(12, 50)  # channel=12 frequency=50Hz
p.start(0)
try:
```

```
    while 1:
        for dc in range(0, 101, 5):
            p.ChangeDutyCycle(dc)
            time.sleep(0.1)
        for dc in range(100, -1, -5):
            p.ChangeDutyCycle(dc)
            time.sleep(0.1)
except KeyboardInterrupt:
    pass
p.stop()
GPIO.cleanup()
```

8.7 检查 GPIO 通道功能

gpio_function(channel) 显示 GPIO channel 的功能：

```
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BOARD)
func = GPIO.gpio_function(pin)
```

返回值包括 GPIO.IN、GPIO.OUT、GPIO.SPI、GPIO.I2C、GPIO.HARD_PWM、GPIO.SERIAL 和 GPIO.UNKNOWN。

参考文献

[1] Mark Lutz, Learning Python, Fourth Edition,O'Reilly Media, Inc., 2009

[2] Raspberry Pi official website, https://www.Raspberrypi.org,Raspberry PI Foundation

[3] Embedded Linux, https://www.elinux.org,Embedded Linux

[4] GPIO source code, https://sourceforge.net/projects/raspberry-gpio-python/,sourceforge

[5] GPIO WIKI, https://sourceforge.net/p/raspberry-gpio-python/wiki/Home,wiki