器称为pip。pip使得为Python安装社区开发软件库变得容易，它常常被Python开发人员使用。

9.10 面向对象的编程

编程语言被设计来支持特定的编程范例或方法，例如过程式编程、函数式编程和面向对象的编程。一种语言可以被设计为支持一个或多个范例，软件开发人员用适合某种范例的方法来使用该语言。让我们来看一个流行的范例：面向对象的编程。这是一种编程方法，其中的代码和数据通过称为"对象"的结构组合在一起。对象的意思是以模拟现实世界概念的方式来表示数据和功能的逻辑分组。

面向对象的编程语言通常使用基于类的方法。"类"是对象的蓝图。从类创建的对象被称为该类的一个"实例"。类中的函数定义被称为"方法"，类中声明的变量被称为"字段"。在Python中，类中每个实例具有不同值的字段被称为"实例变量"，而类中所有实例都有相同值的字段被称为"类变量"。

例如，可以编写一个描述银行账户的类。银行账户类可能有一个表示余额的字段、一个表示持有者姓名的字段，以及用于取款和存款的方法。这个类描述了通用银行账户，但是没有银行账户的具体实例存在，直到从这个类创建一个银行账户对象，如图9-6所示。

如图9-6所示，BankAccount类描述了银行账户的字段和方法，使我们了解银行账户的样子。有两个对象已经被创建，它们是BankAccount类的实例。这些对象是具体的银行账户，分配了姓名和余额。我们可以用每个对象的withdraw和deposit方法来修改其balance字段。在Python中，向名为myAccount的银行账户存款如下，结果是使其balance字段增加25：

```
myAccount.deposit(25)
```

注意

参阅设计18尝试用Python来实现刚才描述的银行账户类。

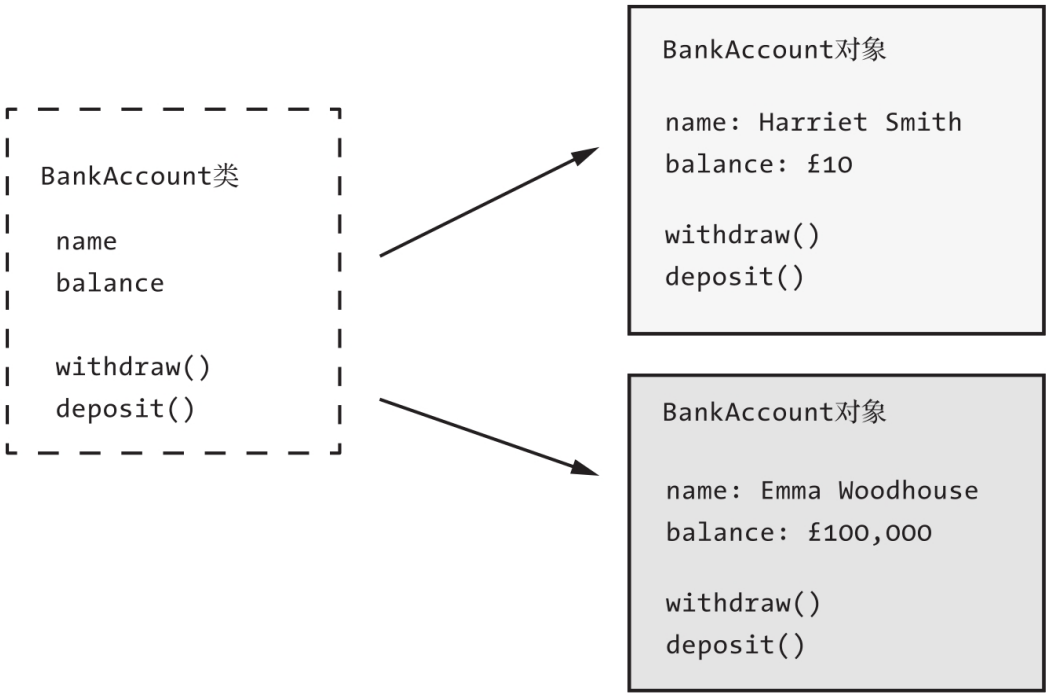

图9-6 从银行账户类创建的银行账户对象

9.11 编译或解释

如前所述，源代码最初是由开发人员编写的程序文本，一般不用CPU能直接理解的编程语言来编写。CPU只能理解机器语言，所以需要额外的步骤：源代码必须编译成机器语言或者在运行时由其他代码解释。

在编译语言（例如C语言）中，源代码被转换成能被处理器直接执行的机器指令。这个过程在9.1节已经描述过了。源代码在开发过程中进行编译，编译后的可执行文件（有时称为二进制文件）被交付给最终用户。当最终用户执行这个二进制文件时，他们无须访问源代码。编译后的代码往往执行速度很快，但它只在编译它的架构上运行。图9-7的例子展示了开发人员如何利用GNU C编译器（gcc）从命令行编译并运行C程序。

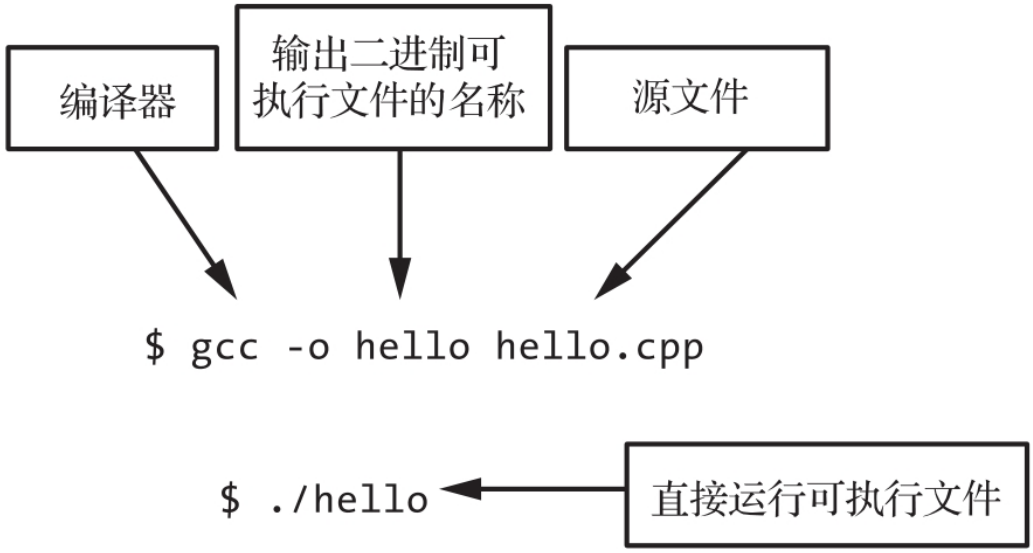

图9-7 把C源文件编译成可独立运行的可执行文件

在解释语言（例如Python语言）中，源代码不会提前编译。相反，它由所谓的解释器程序读取，解释器读取并执行程序的指令。CPU运行的实际上是解释器的机器码。解释语言代码的开发人员可以发布他们的源代码，且最终用户可以直接运行该代码，不需要潜在的复杂编译步骤。在这种情况下，开发人员不用操心如何针对许多不同平台来编译代码——只要用户自己的系统中有合适的解释器，他们就能运行代码。这样，发布的代码就是与平台无关的。

由于运行时解释代码的开销，解释代码往往比编译代码运行得慢。当用户已经安装了所需的解释器，或者用户在技术上足够熟练以至于安装解释器不成问题时，发布解释代码效果最好。否则，开发人员需要把解释器与他们的软件绑定在一起，或者指导用户安装解释器。图9-8展示了从命令行运行Python程序的例子，这里假设已经安装了Python 3解释器。请注意hello.py中的Python源代码是如何直接提供给解释器的——不需要中间步骤。

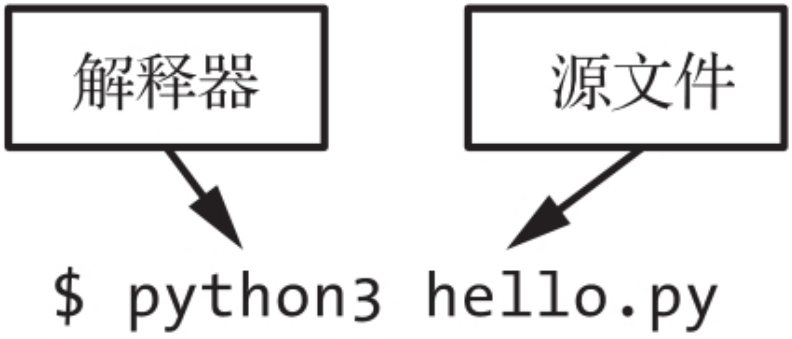

图9-8 Python解释器运行Python源代码

有些语言采用的系统利用这两种方法的混合方法。这样的语言会编译成中间语言或字节码。字节码类似于机器码，但并不针对特定硬件架构，字节码被设计成在虚拟机上运行，如图9-9所示。

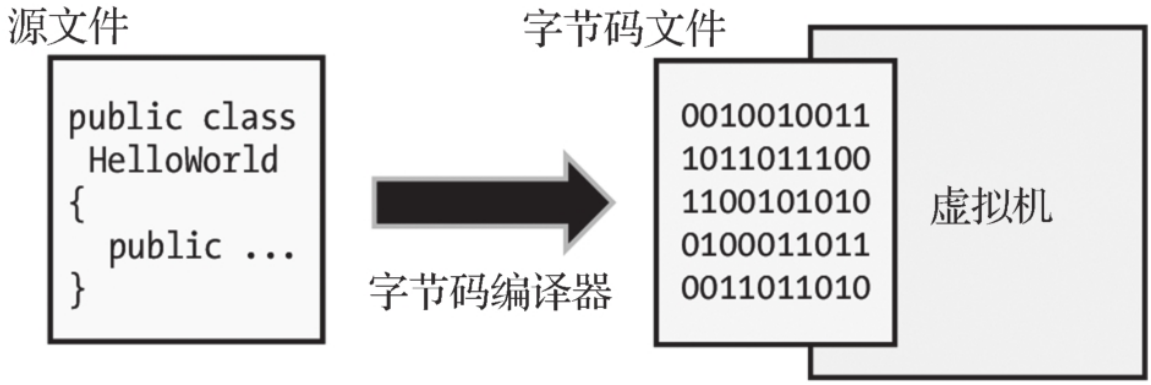

图9-9 字节码编译器把源代码转换成在虚拟机内运行的字节码

在这里，虚拟机是一种用来运行其他软件的软件平台。虚拟机提供虚拟CPU和执行环境，抽象了真实底层硬件和操作系统的详细信息。例如，Java源代码被编译成Java字节码，然后在Java虚拟机中运行。类似地，C#源代码被编译成公共中间语言（Common Intermediate Language，CIL）并在.NET公共语言运行时（Common Language Runtime，CLR）虚拟机中运行。Python的原始实现CPython实际上会在运行Python源代码之前就把它转换成字节码，尽管这是CPython编译器的实现细节，且对绝大多数Python开发人员隐藏。使用字节码的编程语言保留了解释语言的平台独立性，同时也保留了编译代码的一些性能提升。

9.12 用C语言计算阶乘

为了总结高级编程，我们来看看阶乘算法的实现，这次使用C语言。我们之前是用ARM汇编语言实现的，所以用C语言实现同样的逻辑应该可以很好地比较汇编语言和高级语言。这里的C语言代码使用了之前讨论的一些概念。选择使用C语言而不是Python的原因是，C语言是编译语言，我们可以查看编译后的机器码。下面是一个简单的计算数字阶乘的C语言函数：

```
// Calculate the factorial of n.
int factorial(int n)
{
  int result = n;

  while(--n > 0)
  {
    result = result * n;
  }

  return result;
}
```

其他代码可以调用这个函数，传递参数n为要计算其阶乘的数字。该函数在内部计算阶乘值，将其保存到局部变量result中，并把计算值返回给调用者。和在第8章对汇编代码所做的一样，让我们再次用练习来深入研究一下这段代码。

练习9-2：在脑海中运行一个C程序

尝试在脑海中或用纸笔来运行前面的阶乘函数。假设输入值为n=4。当函数返回时，返回的预期值应为24。建议在执行每行代码前后都跟踪一下n和result的值。完成整个代码，直到到达while循环的末尾，看看是否得到了预期值。

注意，while循环的条件（--n>0）把递减运算符（--）放在变量n的前面，这表示在n与0进行比较之前其值需要减1。每次评估while循环条件时都会执行这个操作。

我希望你能发现这个算法的C语言版比ARM汇编版更具可读性。这个版本阶乘代码的另一个主要优点在于它没有与特定类型的处理器绑定。给定合适的编译器，它就可以在所有处理器上编译。如果在ARM处理器上编译早期的C代码，你会看到生成的机器码类似于之前查看的ARM汇编代码。你将有机会在设计19中进行实践，不过这里我已经为你编译并反汇编好了代码：

```
Address   Assembly
0001051c  sub     r3, r0, #1
00010520  cmp     r3, #0
00010524  bxle    lr
00010528  mul     r0, r3, r0
0001052c  subs    r3, r3, #1
00010530  bne     00010528
00010534  bx      lr
```

正你所见，从C语言源程序生成的代码与第8章中介绍的汇编阶乘代码非常相似。虽然也有一些差异，但其具体细节与我们所讨论内容无关。这里需要注意的是，程序可以用像C这样的高级语言编写，编译器可以完成把高级语言语句转换成机器码的困难工作。你可以看到高级语言是如何简化开发人员的工作的，但最终，我们仍然要转换成机器码字节，因为这才是处理器所需要的。

注意

请参阅设计19尝试编译并反汇编C语言的阶乘程序。

这里发生了一些有趣的事情，我想确保你没有错过它。我们从用C编程语言编写的源代码开始，把它编译成机器码，然后再反汇编成汇编语言。这意味着，如果你的计算机上有已经编译好的程序或软件库，你就能把它的代码当作汇编语言代码来查看。你可能无法访问原始源代码，但程序的汇编版本在你的掌握中。

我们一直在专门研究适合ARM处理器的机器码和汇编语言，但就像之前所说的那样，用像C语言那样的高级语言来开发程序的一个好处是，同样的代码可以在不同的处理器上编译。实际上，相同的代码甚至能在不同的操作系统上编译，只要相关代码不使用针对特定操作系统的功能。为了说

明这一点，我已经为32位x86处理器编译了同样的阶乘运算C代码，这次使用的是Windows而不是Linux。下面是生成的机器码，显示为汇编语言：

```
Address   Assembly
00406c35  mov     ecx,dword ptr [esp+4]
00406c39  mov     eax,ecx
00406c3b  jmp     00406c40
00406c3d  imul    eax,ecx
00406c40  dec     ecx
00406c41  test    ecx,ecx
00406c43  jg      00406c3d
00406c45  ret
```

我不会详细介绍这段代码，但你可以随意地研究x86指令集并自行解释该代码。我希望你从这个例子中学到：像C语言这样的高级语言使开发人员编写的代码比汇编语言更易懂，且能轻松地在各种处理器上进行编译。

9.13 总结

本章介绍了高级编程语言。这些语言独立于特定的CPU，在语法上更接近人类语言。你学习了这些语言中常见的元素，比如注释、变量、函数和循环功能。你看到了这些元素是如何用两种编程语言（C语言和Python）表示的。我们查看了一个C语言示例程序，你看到了通过编译高级代码生成的反汇编机器码。

第10章将介绍操作系统。我们将概述操作系统提供的功能，介绍操作系统的各种系列，并深入研究操作系统的工作原理。在此过程中，你将有机会探索Raspberry Pi OS，它是为Raspberry Pi量身定制的Linux版本。

设计14：查看变量

前提条件：运行Raspberry Pi OS的Raspberry Pi。

在本设计中，将编写使用变量的高级代码，并查看它在内存中的工作方式。在主文件夹根目录下，使用文本编辑器创建一个新文件vars.c。在文

本编辑器中输入如下C代码（不需要保留缩进和空行，但要保证保留换行符）：

```
#include <stdio.h>❶
#include <signal.h>

int main()❷
{
  int points = 27;❸
  int year = 2020;❹

  printf("points is %d and is stored at 0x%08x\n", points, &points);❺
  printf("year is %d and is stored at 0x%08x\n", year, &year);

  raise(SIGINT);❻

  return 0;
}
```

现在，让我们查看一下源代码。源代码首先包含了几个头文件❶。这些文件包括了C编译器所需的详细信息，这些信息与程序稍后要使用的printf和raise函数有关。接下来是main函数❷，这是程序开始执行的入口点。程序声明了两个整数变量points❸和year❹，并为它们分配了值。之后，程序输出这两个变量的值和它们的内存地址（以十六进制的形式）❺。raise（SIGINT）语句使程序停止执行❻。通常，最终用户运行的代码中不使用该语句，它是用于辅助调试的一种技术。

文件保存后，用GNU C编译器（gcc）把代码编译成可执行文件。在Raspberry Pi上打开一个终端，输入如下命令来调用编译器。这个命令把vars.c作为输入，编译并链接代码，然后输出一个名为vars的可执行文件。

```
$ gcc -o vars vars.c
```

现在尝试用如下命令运行已编译的代码。程序应输出其两个变量的值和地址：

```
$ ./vars
```

确认程序正常工作后，就在GNU调试器（gdb）下运行该程序，查看内存中的变量：

```
$ gdb vars
```

此时gdb已经加载了文件，但指令都还未执行。在（gdb）提示符下，输入如下内容开始执行程序，该程序会一直执行，直到执行raise（SIGINT）语句。

```
(gdb) run
```

当程序返回（gdb）提示符后，你应该会看到输出的变量值和内存地址的信息。在这些信息后面，你可能会看到一条类似“no such file or directory”这样令人担忧的语句——你可以忽略它。它只表明调试器试图寻找一些不在系统上的源代码。需要注意的是，输出应该是这样的：

```
Starting program: /home/pi/vars
points is 27 and is stored at 0x7efff1d4
year is 2020 and is stored at 0x7efff1d0
```

现在你知道了内存地址，而且由于你已经在使用调试器了，因此很容易查看这些地址中保存的内容。在这个输出中，你可以看到year保存在低地址，points保存在4个字节之后，所以你将从year变量的地址（在本例中为0x7efff1d0）开始转储内存。下面的命令从0x7efff1d0地址开始，以十六进制的形式转储了内存中的3个32位值。你的地址可能不同，如果不同，请用你系统上的year地址替换0x7efff1d0。

```
(gdb) x/3xw 0x7efff1d0
0x7efff1d0:     0x000007e4      0x0000001b      0x00000000
```

在这里，你可以看到存储在0x7efff1d0中的值为0x000007e4。这是十进制的2020，即预期的year值。4个字节后存储的值为0x0000001b，即十进制的27，也是预期的points值。内存中的下一个值正好是0，它不是我们的变量。我们通常用十六进制来查看内存，但如果你想用十进制查看这些值，则可以改用如下命令：

```
(gdb) x/3dw 0x7efff1d0
0x7efff1d0:     2020    27      0
```

你正在查看32位（4字节）内存块，因为这是这个程序使用的变量大小。但内存实际上是按字节编址的，这意味着每个字节都有自己的地址。这就是points的地址比year的地址大4个字节的原因。我们来看看相同内存范围中的字节序列：

```
(gdb) x/12xb 0x7efff1d0
0x7efff1d0:     0xe4    0x07    0x00    0x00    0x1b    0x00    0x00    0x00
0x7efff1d8:     0x00    0x00    0x00    0x00
```

这里重点看一下year的值，注意一下最低有效字节（0xe4）为什么排在第一个。这是由于采用的是小端序存储（设计13讨论过）。你可以用q退出gdb（即使调试会话还是活跃的，它也会问你是否想要退出，回答y即可）。

设计15：改变Python中变量引用的值类型

前提条件：运行Raspberry Pi OS的Raspberry Pi。

在本设计中，你将编写代码把Python变量设置成某种类型的值，然后再更新该变量以引用不同类型的值。在主文件夹根目录下，使用文本编辑器创建一个新文件vartype.py。在文本编辑器中输入如下Python代码：

```
age = 22
print('What is the type?')
print(type(age))

age = 'twenty-two'
print('Now what is the type?')
print(type(age))
```

这段代码为名为age的变量设置一个整数值并输出该值的类型，然后再为age设置字符串值并再次输出其类型。

保存文件后，你可以用Python解释器从终端窗口运行该文件，如下所示：

```
$ python3 vartype.py
```

你应该看到如下输出：

```
What is the type?
<class 'int'>
Now what is the type?
<class 'str'>
```

你可以看到如何仅通过给变量赋新值就能把值类型从整数修改为字符串。不要被术语class迷惑了，在Python 3中，内置类型（比如int和str）被看作类（9.10节有介绍）。在Python中为变量设置不同类型的值很容易，但这在C语言中是根本不被允许的。

Python的版本

现在Python有两个主要的版本正在使用中，即Python 2和Python 3。自2020年1月1日起Python 2不再得到支持，这意味着Python 2将不会有新的漏洞修复。Python开发人员被鼓励把之前的项目迁移到Python 3，且新项目应该采用Python 3。因此，本书中的设计项目使用Python 3。在Raspberry Pi OS以及其他一些Linux发行版上，从命令行运行python将会调用Python 2解释器，而运行python3将会调用Python 3解释器。这就是本书中的设计项目特地要求你运行python3而不是python的原因。也就是说，在其他平台上，甚至是在未来的Raspberry Pi OS版本中，这可能是不适用的，且输入python可能实际调用Python 3。你可以按如下方法检查调用的Python版本：

```
$ python --version
$ python3 --version
```

设计16：栈或堆

前提条件：设计14。

在本设计中，你将在正在运行的程序中查看变量在栈或堆中是否分配了内存空间。在Raspberry Pi上打开一个终端，开始调试之前在设计14中编译的vars程序：

```
$ gdb vars
```

此时，gdb已经加载了文件，但指令还没有执行。在gdb提示符下，输入如下内容以执行程序，该程序会一直执行，直到执行SIGINT语句：

```
(gdb) run
```

再次查看points和year变量的内存地址。在我的代码中，这些变量位于0x7efff1d4和0x7efff1d0，但你的地址可能不同。现在，用如下命令来查看正在运行的程序的全部内存映射位置：

```
(gdb) info proc mappings
```

输出列出了这个程序使用的各种内存范围的开始和结束地址。找到包含变量地址的那个。这两个变量地址应该在一个范围内。对我而言，这一项匹配：

```
0x7efdf000 0x7f000000    0x21000        0x0 [stack]
```

如你所见，gdb表明这个内存范围被分配给了栈，这正好是我们期望局部变量应该在的位置。你可以用q退出gdb（即使调试会话还是活跃的，它也会问你是否想要退出，回答y即可）。

现在，让我们来看看堆上分配的内存。你需要修改vars.c并重新构建它以便程序分配一些堆内存。用文本编辑器打开现有的vars.c文件。在首行添加下面的代码行：

```
#include <stdlib.h>
```

然后在SIGINT行前面添加下面的两行代码：

```
void * data = malloc(512);
printf("data is 0x%08x and is stored at 0x%08x\n", data, &data);
```

我们来介绍一下这些改变的含义。我们调用内存分配函数malloc从堆中分配512字节的内存。malloc函数返回新分配内存的地址。这个地址保存在

名为data的新局部变量中。然后，程序输出两个内存地址：新的堆分配地址和data变量自身的地址（它应该在栈上）。

保存文件后，用gcc编译代码：

```
$ gcc -o vars vars.c
```

现在，再次运行这个程序：

```
$ gdb vars
(gdb) run
```

检查新输出的值。对我而言，输出的值如下：

```
data is 0x00022410 and is stored at 0x7efff1ac
```

我们希望第一个地址（即从malloc返回的地址）在堆上。第二个地址（即data局部变量的地址）应该在栈上。同样，运行如下命令以查看该程序的内存范围以及这两个地址的位置。

```
(gdb) info proc mappings

...
          0x22000    0x43000    0x21000        0x0 [heap]
...
       0x7efdf000 0x7f000000    0x21000        0x0 [stack]
```

找到匹配的地址范围，并确认该地址落在预期的堆和栈范围中。你可以用q退出gdb。

设计17：编写猜谜游戏

在本设计中，将以本章介绍的内容为基础，用Python编写一个猜谜游戏。在主文件夹根目录下，使用文本编辑器创建一个新文件guess.py。在文本编辑器中输入如下Python代码。在Python中，缩进很重要，所以请确保采用合适的缩进。

```
from random import randint❶

secret = randint(1, 10)❷
guess = 0❸
count = 0❹

print('Guess the secret number between 1 and 10')

while guess != secret:❺
    guess = int(input())❻
    count += 1

    if guess == secret:❼
        print('You got it! Nice job.')
    elif guess < secret:
        print('Too low. Try again.')
    else:
        print('Too high. Try again.')

print('You guessed {0} times.'.format(count))❽
```

我们来看看这个程序是如何工作的。这段代码首先导入一个名为randint的函数，该函数产生随机整数❶。这是使用别人编写的函数的例子，randint是Python标准库的一部分。对randint的调用返回1～10的一个随机整数，我们把它存储在名为secret的变量中❷。然后，代码把变量guess设置为0❸。这个变量保存玩家猜测的数字，其初始值为0，我们可以肯定这个值与secret的值不匹配。第三个变量名为count❹，它跟踪玩家到目前为止猜测的次数。

只要玩家猜测的数字与secret不匹配就继续运行while循环❺。循环内的代码调用内置函数input，从控制台获取用户猜测的数字❻，将其转换成整数并保存到guess变量中。每次输入猜测数字时，都要把它与secret变量进行比较，看它们是否匹配❼。当玩家的guess与secret匹配时，退出循环，程序输出玩家猜测的次数❽。

保存文件后，你可以用Python解释器运行该文件：

```
$ python3 guess.py
```

多运行几次这个程序，每次运行时都修改secret的值。你可能想试着修改程序，以便扩大允许的整数范围或者加入你自己的自定义消息。作为挑

战，请尝试修改程序，使得在猜测数字非常接近正确值时，程序输出不同的消息。

设计18：使用Python中的银行账户类

在本设计中，你将用Python编写一个银行账户类，然后以该类为基础创建一个对象。在主文件夹根目录下，使用文本编辑器创建一个新文件bank.py。在文本编辑器中输入如下Python代码。如果你愿意，可以不输入注释（以#开头的行）。注意__init__的开始和结尾有两个下划线。

```
# Define a bank account class in Python.
class BankAccount:❶
    def __init__(self, balance, name):❷
        self.balance = balance❸
        self.name = name❹

    def withdraw(self, amount):❺
        self.balance = self.balance - amount

    def deposit(self, amount):❻
        self.balance = self.balance + amount

# Create a bank account object based on the class.
smithAccount = BankAccount(10.0, 'Harriet Smith')❼

# Deposit some additional money to the account.
smithAccount.deposit(5.25)❽


# Print the account balance.
print(smithAccount.balance)❾
```

这段代码定义了一个名为BankAccount的新类❶。当创建该类的实例时，自动调用其__init__函数❷。这个函数把实例变量balance❸和name❹设置为传递给初始化函数的值。这些变量对于该类创建的每个对象实例而言都是独一无二的。类定义中还包含两个方法：withdraw❺和deposit❻。它们只修改balance。定义类之后，便可创建类的实例❼。现在可以通过访问其变量和方法来使用这个银行账号对象。这里先进行存款❽，之后检索新余额并输出❾。

保存文件后，你可以用Python解释器运行该文件：