第 7 章 函数

内容提要

在本章中，你将学习编写函数。函数是带名字的代码块，用于完成具体的工作。要执行函数定义的特定任务，可调用该函数。需要在程序中多次执行同一项任务时，无须反复编写完成该任务的代码，只需要调用执行该任务的函数，让 Python 运行其中的代码即可。你将发现，通过使用函数，程序编写、阅读、测试和修复起来都更加容易。

7.1 定义函数

下面是一个打印问候语的简单函数，名为 greet_user()： greeter.py

```
def greet_user():
"""显示简单的问候语。"""
print("Hello!")
greet_user()
```

7.1.1 向函数传递信息

只需稍作修改，就可让函数 greet_user() 不仅向用户显示 Hello! ，还将用户的名字作为抬头。为此，可在函数定义 def greet_user() 的括号内添加 username 。通过在这里添加 username ，可让函数接受你给 username 指定的任何值。现在，这个函数要求你调用它时给 username 指定一个值。调用 greet_user() 时，可将一个名字传递给它，

```
def greet_user(username):
    """显示简单的问候语。"""
    print(f"Hello, {username.title()}!")
greet_user('jesse')
```

7.1.2 实参和形参

前面定义函数 greet_user() 时，要求给变量 username 指定一个值。调用这个函数并提供这种信息（人名）时，它将打印相应的问候语。

在函数 greet_user() 的定义中，变量 username 是一个形参（parameter），即函数完成工作所需的信息。在代码 greet_user('jesse') 中，值'jesse' 是一个实参（argument），即调用函数时传递给函数的信息。调用函数时，将要让函数使用的信息放在圆括号内。在 greet_user('jesse') 中，将实参'jesse' 传递给了函数 greet_user() ，这个值被赋给了形参 username 。

7.2 传递实参

函数定义中可能包含多个形参，因此函数调用中也可能包含多个实参。向函数传递实参的方式很多：可使用位置实参，这要求实参的顺序与形参的顺序相同；也可使用关键字实参，其中每个实参都由变量名和值组成；还可使用列表和字典。

7.2.1 位置实参

调用函数时，Python 必须将函数调用中的每个实参都关联到函数定义中的一个形参。为此，最简单的关联方式是基于实参的顺序。这种关联方式称为位置实参。

为明白其中的工作原理，来看一个显示宠物信息的函数。这个函数指出一个宠物属于哪种动物以及它叫什么名字

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息。"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet('hamster', 'harry')
```

7.2.2 默认值

编写函数时，可给每个形参指定默认值。在调用函数中给形参 供了实参时，Python 将使用指定的实参值；否则，将使用形参的默认值。因此，给形参指定默认值后，可在函数调用中省略相应的实参。使用默认值可简化函数调用，还可清楚地指出函数的典型用法。

如果你发现调用 describe_pet() 时，描述的大多是小狗，就可将形参 animal_type 的默认值设置为'dog' 。这样，调用 describe_pet() 来描述小狗时，就可不提供这种信息：

```
def describe_pet(pet_name, animal_type='dog'):
    """显示宠物的信息。"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet(pet_name='willie')
```

这里修改了函数 describe_pet() 的定义，在其中给形参 animal_type 指定了默认值'dog' 。这样，调用这个函数时，如果没有给 animal_type 指定值，Python 就将把这个形参设置为'dog' 。

第 7 章 练习

1. 编写一个名为 $\text{make}_s hirt()TTT$

2. 修改函数 $\text{make}_s hirt()IlovePythonTTTT$

7.3 返回值

函数并非总是直接显示输出，它还可以处理一些数据，并返回一个或一组值。函数返回的值称为返回值。在函数中，可使用 return 语句将值返回到调用函数的代码行。返回值让你能够将程序的大部分繁重工作移到函数中去完成，从而简化主程序。

7.3.1 返回简单值

formatted_name.py

```
def get_formatted_name(first_name, last_name):
    """返回整洁的姓名。"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()
musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

7.3.2 让实参变成可选的

有时候，需要让实参变成可选的，这样使用函数的人就能只在必要时提供额外的信息。可使用默认值来让实参变成可选的。

```
def get_formatted_name(first_name, middle_name, last_name):
    """返回整洁的姓名。"""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()
musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

并非所有的人都有中间名，但如果调用这个函数时只 供了名和姓，它将不能正确运行。为了让中间名变成可选的，可给形参 middle_name 指定一个空的默认值，并在用户没有 供中间名时不使用这个形参。为让 get_formatted_name() 在没有 供中间名时依然可行，可将形参 middle_name 的默认值设置为空字符串，并将其移到形参列表的末尾：

```
def get_formatted_name(first_name, last_name, middle_name=''):

    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()
musician = get_formatted_name('jimi', 'hendrix')
print(musician)
musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

可选值让函数能够处理各种不同的情形，同时确保函数调用尽可能简单。

7.3.3 返回字典

函数可返回任何类型的值，包括列表和字典等较复杂的数据结构。例如，下面的函数接受姓名的组成部分，并返回一个表示人的字典：

person.py

```
def build_person(first_name, last_name):
    """返回一个字典，其中包含有关一个人的信息。"""
    person = {'first': first_name, 'last': last_name}
    return person
musician = build_person('jimi', 'hendrix')
print(musician)
```

这个函数接受简单的文本信息，并将其放在一个更合适的数据结构中，让你不仅能打印这些信息，还能以其他方式处理它们。当前，字符串'jimi' 和'hendrix' 被标记为名和姓。你可以轻松地扩展这个函数，使其接受可选值，如中间名、年龄、职业或其他任何要存储的信息。

```
def build_person(first_name, last_name, age=None):
    """返回一个字典，其中包含有关一个人的信息。"""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person
musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

在函数定义中，新增了一个可选形参 age ，并将其默认值设置为特殊值 None （表示变量没有值）。可将 None 视为占位值。在条件测试中，None 相当于 False 。如果函数调用中包含形参 age 的值，这个值将被存储到字典中。在任何情况下，这个函数都会存储人的姓名，但可进行修改，使其同时存储有关人的其他信息。

7.4 传递列表

你经常会发现，向函数传递列表很有用，其中包含的可能是名字、数或更复杂的对象（如字典）。将列表传递给函数后，函数就能直接访问其内容。下面使用函数来 高处理列表的效率。

假设有一个用户列表，我们要问候其中的每位用户。下面的示例将包含名字的列表传递给一个名为 greet_users() 的函数，这个函数问候列表中的每个人：

greet_users.py

```
def greet_users(names):
    """向列表中的每位用户发出简单的问候。"""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)
usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

7.4.1 在函数中修改列表

将列表传递给函数后，函数就可对其进行修改。在函数中对这个列表所做的任何修改都是永久性的，这让你能够高效地处理大量数据。

来看一家为用户提交的设计制作 3D 打印模型的公司。需要打印的设计存储在一个列表中，打印后将移到另一个列表中。下面是在不使用函数的情况下模拟这个过程的代码：

printing_models.py

```
# 首先创建一个列表，其中包含一些要打印的设计。
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# 模拟打印每个设计，直到没有未打印的设计为止。
# 打印每个设计后，都将其移到列表completed_models中。
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# 显示打印好的所有模型。
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

这个程序首先创建一个需要打印的设计列表，以及一个名为 completed_models 的空列表，每个设计打印后都将移到其中。只要列表 unprinted_designs 中还有设计，while 循环就模拟打印设计的过程：从该列表末尾删除一个设计，将其赋给变量 current_design ，并显示一条消息指出正在打印当前的设计，然后将该设计加入到列表 completed_models 中。循环结束后，显示已打印的所有设计。

为重新组织这些代码，可编写两个函数，每个都做一件具体的工作。大部分代码与原来相同，只是效率更高。第一个函数负责处理打印设计的工作，第二个概述打印了哪些设计：

```
def print_models(unprinted_designs, completed_models):
    """
    模拟打印每个设计，直到没有未打印的设计为止。
    打印每个设计后，都将其移到列表completed_models中。
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)


def show_completed_models(completed_models):
    """显示打印好的所有模型。"""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)


unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
```

```
print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

第 7 章 练习

1. 创建一个列表,其中包含一系列简短的文本消息。将该列表传递给一个名为 show_messages() 的函数，这个函数会打印列表中的每条文本消息。
2. 编写一个名为 send_messages() 的函数,将每条消息都打印出来并移到一个名为 sent_messages 的列表中。调用函数 send_messages() ，再将两个列表都打印出来，确认正确地移动了消息。

7.5 传递任意数量的实参

有时候，预先不知道函数需要接受多少个实参，好在 Python 允许函数从调用语句中收集任意数量的实参。

例如，来看一个制作比萨的函数，它需要接受很多配料，但无法预先确定顾客要多少种配料。下面的函数只有一个形参 *toppings ，但不管调用语句 供了多少实参，这个形参会将它们统统收入囊中：

pizza.py

```
def make_pizza(*toppings):
    """打印顾客点的所有配料。"""
    print(toppings)
make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

形参名 *toppings 中的星号让 Python 创建一个名为 toppings 的空元组，并将收到的所有值都封装到这个元组中。函数体内的函数调用 print() 通过生成输出，证明 Python 能够处理使用一个值来调用函数的情形，也能处理使用三个值来调用函数的情形。它以类似的方式处理不同的调用。

注 Python 将实参封装到一个元组中，即便函数只收到一个值

可以将函数调用 print() 替换为一个循环，遍历配料列表并对顾客点的比萨进行描述：

```
def make_pizza(*toppings):
    """概述要制作的比萨。"""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

不管函数收到的实参是多少个，这种语法都管用。

7.5.1 结合使用位置实参和任意数量实参

如果要让函数接受不同类型的实参，必须在函数定义中将接纳任意数量实参的形参放在最后。Python 先匹配位置实参和关键字实参，再将余下的实参都收集到最后一个形参中。

例如，如果前面的函数还需要一个表示比萨尺寸的形参，必须将其放在形参 *toppings 的前面

```
def make_pizza(size, *toppings):
    """概述要制作的比萨。"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

基于上述函数定义，Python 将收到的第一个值赋给形参 size ，并将其他所有值都存储在元组 toppings 中。在函数调用中，首先指定表示比萨尺寸的实参，再根据需要指定任意数量的配料。

7.5.2 使用任意数量的关键字实参

有时候，需要接受任意数量的实参，但预先不知道传递给函数的会是什么样的信息。在这种情况下，可将函数编写成能够接受任意数量的键值对——调用语句提供了多少就接受多少。一个这样的示例是创建用户简介：你知道将收到有关用户的信息，但不确定会是什么样的信息。在下面的示例中，函数 build_profile() 接受名和姓，还接受任意数量的关键字实参：

user_profile.py

```
def make_pizza(size, *toppings):
    """概述要制作的比萨。"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

基于上述函数定义，Python 将收到的第一个值赋给形参 size ，并将其他所有值都存储在元组 toppings 中。在函数调用中，首先指定表示比萨尺寸的实参，再根据需要指定任意数量的配料。

现在，每个比萨都有了尺寸和一系列配料，而且这些信息按正确的顺序打印出来了——首先是尺寸，然后是配料：

```
Making a 16-inch pizza with the following toppings:
- pepperoni
Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

注 你经常会看到通用形参名 *args ，它也收集任意数量的位置实参。

7.5.3 使用任意数量的关键字实参

有时候，需要接受任意数量的实参，但预先不知道传递给函数的会是什么样的信息。在这种情况下，可将函数编写成能够接受任意数量的键值对——调用语句提供了多少就接受多少。一个这样的示例是创建用户简介：你知道将收到有关用户的信息，但不确定会是什么样的信息。在下面的示例中，函数 build_profile() 接受名和姓，还接受任意数量的关键字实参：

user_profile.py

```
def build_profile(first, last, **user_info):
    """创建一个字典，其中包含我们知道的有关用户的一切。"""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info
user_profile = build_profile('albert', 'einstein',
location='princeton',
field='physics')
print(user_profile)
```

函数 build_profile() 的定义要求提供名和姓，同时允许根据需要提供任意数量的名称值对。形参 **user_info 中的两个星号让 Python 创建一个名为 user_info 的空字典，并将收到的所有名称值对都放到这个字典中。在这个函数中，可以像访问其他字典那样访问 user_info 中的名称值对。

在 build_profile() 的函数体内，将名和姓加入了字典 user_info 中，因为总是会从用户那里收到这两项信息，而这两项信息没有放到这个字典中。接下来，将字典 user_info 返回到函数调用行。我们调用 build_profile() ，向它传递名（'albert' ）、姓（'einstein' ）和两个键值对（location='princeton' 和 field='physics' ），并将返回的 user_info 赋给变量 user_profile ，再打印该变量：

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

在这里，返回的字典包含用户的名和姓，还有求学的地方和所学专业。调用这个函数时，不管额外 供多少个键值对，它都能正确地处理。

注 你经常会看到形参名 **kwargs ，它用于收集任意数量的关键字实参。

第 7 章 练习

1. 编写一个函数，它接受顾客要在三明治中添加的一系列食材。这个函数只有一个形参（它收集函数调用中提供的所有食材），并打印一条消息，对顾客点的三明治进行概述。调用这个函数三次，每次都 供不同数量的实参。
2. 编写一个函数，将一辆汽车的信息存储在字典中。这个函数总是接受制造商和型号，还接受任意数量的关键字实参。这样调用该函数：提供必不可少的信息，以及两个名称值对，如颜色和选装配件。这个函数必须能够像下面这样进行调用：

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

打印返回的字典，确认正确地处理了所有的信息。

7.6 将函数存储在模块中

使用函数的优点之一是可将代码块与主程序分离。通过给函数指定描述性名称，可让主程序容易理解得多。你还可以更进一步，将函数存储在称为模块的独立文件中，再将模块导入到主程序中。import 语句允许在当前运行的程序文件中使用模块中的代码。

通过将函数存储在独立的文件中，可隐藏程序代码的细节，将重点放在程序的高层逻辑上。这还能让你在众多不同的程序中重用函数。将函数存储在独立文件中后，可与其他程序员共享这些文件而不是整个程序。知道如何导入函数还能让你使用其他程序员编写的函数库。

导入模块的方法有多种，下面对每种进行简要的介绍。

7.6.1 导入整个模块

要让函数是可导入的，得先创建模块。模块是扩展名为.py 的文件，包含要导入到程序中的代码。下面来创建一个包含函数 make_pizza() 的模块。为此，将文件 pizza.py 中除函数 make_pizza() 之外的其他代码删除：

pizza.py

```
def make_pizza(size, *toppings):
    """概述要制作的比萨。"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

接下来，在 pizza.py 所在的目录中创建一个名为 making_pizzas.py 的文件。这个文件导入刚创建的模块，再调用 make_pizza() 两次：

making_pizzas.py

```
import pizza
pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Python 读取这个文件时，代码行 import pizza 让 Python 打开文件 pizza.py，并将其中的所有函数都复制到这个程序中。你看不到复制的代码，因为在这个程序即将运行时，Python 在幕后复制了这些代码。你只需知道，在 making_pizzas.py 中，可使用 pizza.py 中定义的所有函数。

这就是一种导入方法：只需编写一条 import 语句并在其中指定模块名，就可在程序中使用该模块中的所有函数。如果使用这种 import 语句导入了名为 module_name.py 的整个模块，就可使用下面的语法来使用其中任何一个函数：

```
module_name.function_name()
```

7.6.2 导入特定的函数

还可以导入模块中的特定函数，这种导入方法的语法如下：

```
from module_name import function_name
```

通过用逗号分隔函数名，可根据需要从模块中导入任意数量的函数：

```
from module_name import function_0, function_1, function_2
```

7.6.3 使用 as 给函数指定别名

如果要导入函数的名称可能与程序中现有的名称冲突，或者函数的名称太长，可指定简短而独一无二的别名：函数的另一个名称，类似于外号。要给函数取这种特殊外号，需要在导入它时指定。

下面给函数 make_pizza() 指定了别名 mp() 。这是在 import 语句中使用 make_pizza as mp 实现的，关键字 as 将函数重命名为指定的别名：

```
from pizza import make_pizza as mp
mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

上面的 import 语句将函数 make_pizza() 重命名为 mp() 。在这个程序中，每当需要调用 make_pizza() 时，都可简写成 mp() 。Python 将运行 make_pizza() 中的代码，避免与这个程序可能包含的函数 make_pizza() 混淆。

指定别名的通用语法如下：

```
from module_name import function_name as fn
```

7.6.4 使用 as 给模块指定别名

还可以给模块指定别名。通过给模块指定简短的别名（如给模块 pizza 指定别名 p ），让你能够更轻松地调用模块中的函数。相比于 pizza.make_pizza() ， p.make_pizza() 更为简洁：

```
import pizza as p
p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

上述 import 语句给模块 pizza 指定了别名 p ，但该模块中所有函数的名称都没变。要调用函数 make_pizza() ，可编写代码 p.make_pizza() 而非 pizza.make_pizza() 。这样不仅代码更简洁，还让你不用再关注模块名，只专注于描述性的函数名。这些函数名明确指出了函数的功能，对于理解代码而言，比模块名更重要。

给模块指定别名的通用语法如下：

```
import module_name as mn
```

7.6.5 导入模块中的所有函数

使用星号（*）运算符可让 Python 导入模块中的所有函数：

```
from pizza import *
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

import 语句中的星号让 Python 将模块 pizza 中的每个函数都复制到这个程序文件中。由于导入了每个函数，可通过名称来调用每个函数，而无须使用句点表示法。然而，使用并非自己编写的大型模块时，最好不要采用这种导入方法。这是因为如果模块中有函数的名称与当前项目中使用的名称相同，可能导致意想不到的结果：Python 可能遇到多个名称相同的函数或变量，进而覆盖函数，而不是分别导入所有的函数。

最佳的做法是，要么只导入需要使用的函数，要么导入整个模块并使用句点表示法。这让代码更清晰，更容易阅读和理解。这里之所以介绍这种导入方法，只是想让你在阅读别人编写的代码时，能够理解类似于下面的 import 语句：

```
from module_name import *
```

7.7 函数编写指南

编写函数时，需要牢记几个细节。应给函数指定描述性名称，且只在其中使用小写字母和下划线。描述性名称可帮助你和别人明白代码想要做什么。给模块命名时也应遵循上述约定。

每个函数都应包含简要地阐述其功能的注释。该注释应紧跟在函数定义后面，并采用文档字符串格式。文档良好的函数让其他程序员只需阅读文档字符串中的描述就能够使用它。他们完全可以相信代码如描述的那样运行，并且只要知道函数的名称、需要的实参以及返回值的类型，就能在自己的程序中使用它。

给形参指定默认值时，等号两边不要有空格：

```
def function_name(parameter_0, parameter_1='default value'):
```

对于函数调用中的关键字实参，也应遵循这种约定：

```
function_name(value_0, parameter_1='value')
```

第 7 章 练习

1. 将示例 printing_models.py 中的函数放在一个名为 printing_functions.py 的文件中。在 printing_models.py 的开头编写一条 import 语句，并修改该文件以使用导入的函数。
2. 选择一个你编写的且只包含一个函数的程序，将该函数放在另一个文件中。在主程序文件中，使用下述各种方法导入这个函数，再调用它：

```
import module_name
from module_name import function_name
from module_name import function_name as fn
import module_name as mn
from module_name import *
```

第 8 章 类

内容提要

面向对象编程是最有效的软件编写方法之一。在面向对象编程中，你编写表示现实世界中的事物和情景的类，并基于这些类来创建对象。编写类时，你定义一大类对象都有的通用行为。基于类创建对象时，每个对象都自动具备这种通用行为，然后可根据需要赋予每个对象独特的个性。

8.1 创建和使用类

使用类几乎可以模拟任何东西。下面来编写一个表示小狗的简单类 Dog ，它表示的不是特定的小狗，而是任何小狗。对于大多数宠物狗，我们都知道些什么呢？它们都有名字和年龄。我们还知道，大多数小狗还会蹲下和打滚。由于大多数小狗都具备上述两项信息（名字和年龄）和两种行为（蹲下和打滚），我们的 Dog 类将包含它们。这个类让 Python 知道如何创建表示小狗的对象。编写这个类后，我们将使用它来创建表示特定小狗的实例。

8.1.1 创建 Dog 类

根据 Dog 类创建的每个实例都将存储名字和年龄，我们赋予了每条小狗蹲下（sit() ）和打滚（roll_over() ）的能力：dog.py

```
class Dog:
    """一次模拟小狗的简单尝试。"""
    def __init__(self, name, age):
        """初始化属性name和age。"""
        self.name = name
        self.age = age
    def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print(f"{self.name} is now sitting.")
    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print(f"{self.name} rolled over!")
```

方法 __init__()

类中的函数称为方法。你在前面学到的有关函数的一切都适用于方法，就目前而言，唯一重要的差别是调用方法的方式。每当你根据 Dog 类创建新实例时，Python 都会自动运行它。在这个方法的名称中，开头和末尾各有两个下划线，这是一种约定，旨在避免 Python 默认方法与普通方法发生名称冲突。务必确保 __init__() 的两边都有两个下划线，否则当你使用类来创建实例时，将不会自动调用这个方法，进而引发难以发现的错误。

我们将方法 __init__() 定义成包含三个形参：self 、name 和 age 。在这个方法的定义中，形参 self 必不可少，而且必须位于其他形参的前面。为何必须在方法定义中包含形参 self 呢？因

为 Python 调用这个方法来创建 Dog 实例时，将自动传入实参 self 。每个与实例相关联的方法调用都自动传递实参 self ，它是一个指向实例本身的引用，让实例能够访问类中的属性和方法。创建 Dog 实例时，Python 将调用 Dog 类的方法 __init__() 。我们将通过实参向 Dog() 传递名字和年龄，self 会自动传递，因此不需要传递它。每当根据 Dog 类创建实例时，都只需给最后两个形参（name 和 age ）提供值。

以 self 为前缀的变量可供类中的所有方法使用，可以通过类的任何实例来访问。self.name = name 获取与形参 name 相关联的值，并将其赋给变量 name ，然后该变量被关联到当前创建的实例。self.age = age 的作用与此类似。像这样可通过实例访问的变量称为属性。

Dog 类还定义了另外两个方法：sit() 和 roll_over()。这些方法执行时不需要额外的信息，因此它们只有一个形参 self 。我们随后将创建的实例能够访问这些方法，换句话说，它们都会蹲下和打滚。当前，sit() 和 roll_over() 所做的有限，只是打印一条消息，指出小狗正在蹲下或打滚。但可以扩展这些方法以模拟实际情况：如果这个类包含在一个计算机游戏中，这些方法将包含创建小狗蹲下和打滚动画效果的代码；如果这个类是用于控制机器狗的，这些方法将让机器狗做出蹲下和打滚的动作。

8.1.2 根据类创建实例

可将类视为有关如何创建实例的说明。Dog 类是一系列说明，让 Python 知道如何创建表示特定小狗的实例。

下面来创建一个表示特定小狗的实例：

```
class Dog:
    --snip--
my_dog = Dog('Willie', 6)
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
```

这这里使用的是前一个示例中编写的 Dog 类。让 Python 创建一条名字为'Willie' 、年龄为 6 的小狗。遇到这行代码时，Python 使用实参'Willie' 和 6 调用 Dog 类的方法 __init__() 。方法 __init__() 创建一个表示特定小狗的实例，并使用 供的值来设置属性 name 和 age 。接下来，Python 返回一个表示这条小狗的实例，而我们将这个实例赋给了变量 my_dog 。在这里，命名约定很有用：通常可认为首字母大写的名称（如 Dog ）指的是类，而小写的名称（如 my_dog ）指的是根据类创建的实例。

访问属性

要访问实例的属性，可使用句点表示法。

```
my_dog.name
```

句点表示法在 Python 中很常用，这种语法演示了 Python 如何获悉属性的值。在这里，Python 先找到实例 my_dog ，再查找与该实例相关联的属性 name 。在 Dog 类中引用这个属性时，使用的是 self.name 。

调用方法

根据 Dog 类创建实例后，就能使用句点表示法来调用 Dog 类中定义的任何方法了。下面来让小狗蹲下和打滚：

```
class Dog:
    --snip--
my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

要调用方法，可指定实例的名称（这里是 my_dog ）和要调用的方法，并用句点分隔。遇到代码 my_dog.sit() 时，Python 在类 Dog 中查找方法 sit() 并运行其代码。

这种语法很有用。如果给属性和方法指定了合适的 述性名称，如 name 、age 、sit() 和 roll_over() ，即便是从未见过的代码块，我们也能够轻松地推断出它是做什么的。

创建多个实例

可按需求根据类创建任意数量的实例。下面再创建一个名为 your_dog 的小狗实例：

```
class Dog:
    --snip--
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()
print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

在本例中创建了两条小狗，分别名为 Willie 和 Lucy。每条小狗都是一个独立的实例，有自己的一组属性，能够执行相同的操作：

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.
Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

即使给第二条小狗指定同样的名字和年龄，Python 依然会根据 Dog 类创建另一个实例。你可按需求根据一个类创建任意数量的实例，条件是将每个实例都存储在不同的变量中，或者占用列表或字典的不同位置。

第 8 章 练习

1. 创建一个名为 Restaurant 的类，为其方法 __init__() 设置属性 restaurant_name 和 cuisine_type 。创建一个名为 describe_restaurant() 的方法和一个名为 open_restaurant() 的方法，前者打印前述两项信息，而后者打印一条消息，指出餐馆正在营业。
   - 根据这个类创建一个名为 restaurant 的实例，分别打印其两个属性，再调用前述两个方法。

- 编写的类创建三个实例，并对每个实例调用方法 describe_restaurant() 。

8.2 使用类和实例

可使用类来模拟现实世界中的很多情景。类编写好后，你的大部分时间将花在根据类创建的实例上。你需要执行的一个重要任务是修改实例的属性。可以直接修改实例的属性，也可以编写方法以特定的方式进行修改。

8.2.1 Car 类

下面来编写一个表示汽车的类。它存储了有关汽车的信息，还有一个汇总这些信息的方法：

car.py

```
class Car:

    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.make = make
        self.model = model
        self.year = year
    def get_descriptive_name(self):
        """返回整洁的描述性信息。"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name()
```

8.2.2 给属性指定默认值

创建实例时，有些属性无须通过形参来定义，可在方法 __init__() 中为其指定默认值

下面来添加一个名为 odometer_reading 的属性，其初始值总是为 0。我们还添加了一个名为 read_odometer() 的方法，用于读取汽车的里程表：

```
class Car:
    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
def get_descriptive_name(self):
    --snip--
def read_odometer(self):
    """打印一条指出汽车里程的消息。"""
    print(f"This car has {self.odometer_reading} miles on it.")
my_new_car = Car('audi', 'a4', 2019)
```

```
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

8.2.3 修改属性的值

我们能以三种方式修改属性的值：直接通过实例进行修改，通过方法进行设置，以及通过方法进行递增（增加特定的值）。下面依次介绍这些方式。

直接修改属性的值

要修改属性的值，最简单的方式是通过实例直接访问它。下面的代码直接将里程表读数设置为 23：

```
class Car:
    --snip--
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

通过方法修改属性的值

如果有方法能替你更新属性，将大有裨益。这样就无须直接访问属性，而可将值传递给方法，由它在内部进行更新。下面的示例演示了一个名为 update_odometer() 的方法：

```
class Car:
    --snip--
    def update_odometer(self, mileage):
    """将里程表读数设置为指定的值。"""
    self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

通过方法对属性的值进行递增

有时候需要将属性值递增特定的量，而不是将其设置为全新的值。假设我们购买了一辆二手车，且从购买到登记期间增加了 100 英里的里程。下面的方法让我们能够传递这个增量，并相应地增大里程表读数：

```
class Car:
    --snip--
    def update_odometer(self, mileage):
        --snip--
    def increment_odometer(self, miles):
        """将里程表读数增加指定的量。"""
        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2015)
```

```
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23_500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

注 你可以使用类似于上面的方法来控制用户修改属性值（如里程表读数）的方式，但能够访问程序的人都可以通过直接访问属性来将里程表修改为任何值。要确保安全，除了进行类似于前面的基本检查外，还需特别注意细节。

第 8 章 练习

1. 创建一个名为 Restaurant 的类，为其方法 __init__() 设置属性 restaurant_name 和 cuisine_type 。创建一个名为 describe_restaurant() 的方法和一个名为 open_restaurant() 的方法.
   - 添加一个名为 number_served 的属性，并将其默认值设置为 0。根据这个类创建一个名为 restaurant 的实例。打印有多少人在这家餐馆就餐过，然后修改这个值并再次打印它。
   - 添加一个名为 set_number_served() 的方法，让你能够设置就餐人数。调用这个方法并向它传递一个值，然后再次打印这个值。
   - 添加一个名为 increment_number_served() 的方法，让你能够将就餐人数递增。调用这个方法并向它传递一个这样的值：你认为这家餐馆每天可能接待的就餐人数。

8.3 继承

编写类时，并非总是要从空白开始。如果要编写的类是另一个现成类的特殊版本，可使用继承。一个类继承另一个类时，将自动获得另一个类的所有属性和方法。原有的类称为父类，而新类称为子类。子类继承了父类的所有属性和方法，同时还可以定义自己的属性和方法。

8.3.1 子类的方法 __init__()

在既有类的基础上编写新类时，通常要调用父类的方法 __init__() 。这将初始化在父类 __init__() 方法中定义的所有属性，从而让子类包含这些属性。例如，下面来模拟电动汽车。电动汽车是一种特殊的汽车，因此可在前面创建的 Car 类的基础上创建新类 ElectricCar 。这样就只需为电动汽车特有的属性和行为编写代码。下面来创建 ElectricCar 类的一个简单版本，它具备 Car 类的所有功能：

electric_car.py

```
class Car:
    """一次模拟汽车的简单尝试。"""
    def __init__(self, make, model, year):
        self.make = make
```

```
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car):
    """电动汽车的独特之处。"""
    def __init__(self, make, model, year):
        """初始化父类的属性。"""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
```

首先是 Car 类的代码。创建子类时，父类必须包含在当前文件中，且位于子类前面。定义了子类 ElectricCar 。定义子类时，必须在圆括号内指定父类的名称。方法 __init__() 接受创建 Car 实例所需的信息

super() 是一个特殊函数，让你能够调用父类的方法。这行代码让 Python 调用 Car 类的方法 __init__() ，让 ElectricCar 实例包含这个方法中定义的所有属性。父类也称为超类（superclass），名称 super 由此而来。

为测试继承能够正确地发挥作用，我们尝试创建一辆电动汽车，但提供的信息与创建普通汽车时相同。创建 ElectricCar 类的一个实例，并将其赋给变量 my_tesla 。这行代码调用 ElectricCar 类中定义的方法 __init__()，后者让 Python 调用父类 Car 中定义的方法 __init__() 。我们 供了实参'tesla' 、'model s' 和 2019 。

除方法 __init__() 外，电动汽车没有其他特有的属性和方法。当前，我们只想确认电动汽车具备普通汽车的行为

ElectricCar 实例的行为与 Car 实例一样，现在可以开始定义电动汽车特有的属性和方法了。

8.3.2 给子类定义属性和方法

让一个类继承另一个类后，就可以添加区分子类和父类所需的新属性和新方法了。

下面来添加一个电动汽车特有的属性（电瓶），以及一个描述该属性的方法。我们将存储电瓶容量，并编写一个打印电瓶描述的方法：

```
class Car:
    --snip--
class ElectricCar(Car):
    """电动汽车的独特之处。"""
    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery_size = 75
    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

添加了新属性 self.battery_size ，并设置其初始值（75 ）。根据 ElectricCar 类创建的所有实例都将包含该属性，但所有 Car 实例都不包含它。还添加了一个名为 describe_battery() 的方法，打印有关电瓶的信息。调用这个方法时，将看到一条电动汽车特有的描述

对于 ElectricCar 类的特殊程度没有任何限制。模拟电动汽车时，可根据所需的准确程度添加任意数量的属性和方法。如果一个属性或方法是任何汽车都有的，而不是电动汽车特有的，就应将其加入到 Car 类而非 ElectricCar 类中。这样，使用 Car 类的人将获得相应的功能，而 ElectricCar 类只包含处理电动汽车特有属性和行为的代码。

8.3.3 重写父类的方法

对于父类的方法，只要它不符合子类模拟的实物的行为，都可以进行重写。为此，可在子类中定义一个与要重写的父类方法同名的方法。这样，Python 将不会考虑这个父类方法，而只关注你在子类中定义的相应方法

假设 Car 类有一个名为 fill_gas_tank() 的方法，它对全电动汽车来说毫无意义，因此你可能想重写它。下面演示了一种重写方式：

```
class ElectricCar(Car):
    --snip--
    def fill_gas_tank(self):
        """电动汽车没有油箱。"""
        print("This car doesn't need a gas tank!")
```

现在,如果有人对电动汽车调用方法 fill_gas_tank() ,Python 将忽略 Car 类中的方法 fill_gas_tank() ，转而运行上述代码。使用继承时，可让子类保留从父类那里继承而来的精华，并剔除不需要的糟粕。

8.3.4 将实例用作属性

使用代码模拟实物时，你可能会发现自己给类添加的细节越来越多：属性和方法清单以及文件都越来越长。在这种情况下，可能需要将类的一部分 取出来，作为一个独立的类。可以将大型类拆分成多个协同工作的小类。

例如，不断给 ElectricCar 类添加细节时，我们可能发现其中包含很多专门针对汽车电瓶的属性和方法。在这种情况下，可将这些属性和方法提取出来，放到一个名为 Battery 的类中，并将一个 Battery 实例作为 ElectricCar 类的属性：

```
class Car:
    --snip--
class Battery:

    def __init__(self, battery_size=75):
        self.battery_size = battery_size
    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    """电动汽车的独特之处。"""
    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery = Battery()


my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

定义一个名为 Battery 的新类，它没有继承任何类。方法 describe_battery() 也移到了这个类中。

在 ElectricCar 类中，添加了一个名为 self.battery 的属性。这行代码让 Python 创建一个新的 Battery 实例（因为没有指定容量，所以为默认值 75），并将该实例赋给属性 self.battery 。每当方法 __init__() 被调用时，都将执行该操作，因此现在每个 ElectricCar 实例都包含一个自动创建的 Battery 实例。

我们创建一辆电动汽车，并将其赋给变量 my_tesla 。描述电瓶时，需要使用电动汽车的属性 battery ：

```
my_tesla.battery.describe_battery()
```

这行代码让 Python 在实例 my_tesla 中查找属性 battery ，并对存储在该属性中的 Battery 实例调用方法 describe_battery() 。

8.3.5 模拟实物

模拟较复杂的物件（如电动汽车）时，需要解决一些有趣的问题。续航里程是电瓶的属性还是汽车的属性呢？如果只 述一辆汽车，将方法 get_range() 放在 Battery 类中也许是合适的，但如果要描述一家汽车制造商的整个产品线，也许应该将方法 get_range() 移到 ElectricCar 类中。在这种情况下，get_range() 依然根据电瓶容量来确定续航里程，但报告的是一款汽车的续航里程。也可以这样做：仍将方法 get_range() 留在 Battery 类中，但向它传递一个参数，如 car_model 。在这种情况下，方法 get_range() 将根据电瓶容量和汽车型号报告续航里程。

第 8 章 练习

1. 冰激凌小店是一种特殊的餐馆。编写一个名为 IceCreamStand 的类，让它继承之前编写的 Restaurant 类。添加一个名为 flavors 的属性，用于存储一个由各种口味的冰激凌组成的列表。编写一个显示这些冰激凌的方法。创建一个 IceCreamStand 实例，并调用这个方法。
2. 在本节最后一个 electric_car.py 版本中，给 Battery 类添加一个名为 upgrade_battery() 的方法。该方法检查电瓶容量，如果不是 100，就将其设置为 100。创建一辆电瓶容量为默认值的电动汽车，调用方法 get_range() ，然后对电瓶进行升级，并再次调用 get_range() 。你将看到这辆汽车的续航里程增加了。

8.4 导入类

随着不断给类添加功能，文件可能变得很长，即便妥善地使用了继承亦如此。为遵循 Python 的总体理念，应让文件尽可能整洁。Python 在这方面提供了帮助，允许将类存储在模块中，然后在主程序中导入所需的模块。

8.4.1 导入单个类

下面来创建一个只包含 Car 类的模块。这让我们面临一个微妙的命名问题：在本章中已经有一个名为 car.py 的文件，但这个模块也应命名为 car.py，因为它包含表示汽车的代码。我们将这样解决这个命名问题：将 Car 类存储在一个名为 car.py 的模块中，该模块将覆盖前面使用的文件 car.py。从现在开始，使用该模块的程序都必须使用更具体的文件名，如 my_car.py。下面是模块 car.py，其中只包含 Car 类的代码：

car.py

```
class Car:

    def __init__(self, make, model, year):

        self.make = make
```

```
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):

        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """打印一条消息，指出汽车的里程。"""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        将里程表读数设置为指定的值。
        拒绝将里程表往回调。
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """将里程表读数增加指定的量。"""
        self.odometer_reading += miles
```

下面来创建另一个文件 my_car.py，在其中导入 Car 类并创建其实例：

my_car.py

```
from car import Car
    my_new_car = Car('audi', 'a4', 2019)
    print(my_new_car.get_descriptive_name())
    my_new_car.odometer_reading = 23
    my_new_car.read_odometer()
```

import 语句让 Python 打开模块 car 并导入其中的 Car 类。这样，我们就可以使用 Car 类，就像它是在这个文件中定义的一样。

8.4.2 在一个模块中存储多个类

虽然同一个模块中的类之间应存在某种相关性，但可根据需要在一个模块中存储任意数量的类。Battery 类和 ElectricCar 类都可帮助模拟汽车，下面将它们都加入模块 car.py 中：

car.py

```
class Car:
    --snip--
class Battery:

    def __init__(self, battery_size=75):
```

```
        """初始化电瓶的属性。"""
        self.battery_size = battery_size
    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kWh battery.")
    def get_range(self):

        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315
        print(f"This car can go about {range} miles on a full charge.")
class ElectricCar(Car):
    """模拟电动汽车的独特之处。"""
    def __init__(self, make, model, year):

        super().__init__(make, model, year)
        self.battery = Battery()
```

现在，可以新建一个名为 my_electric_car.py 的文件，导入 ElectricCar 类，并创建一辆电动汽车了：

my_electric_car.py

```
from car import ElectricCar
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

8.4.3 从一个模块中导入多个类

可根据需要在程序文件中导入任意数量的类。如果要在同一个程序中创建普通汽车和电动汽车，就需要将 Car 类和 ElectricCar 类都导入：

my_cars.py

```
from car import Car, ElectricCar
my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

8.5 Python 标准库

Python 标准库是一组模块，我们安装的 Python 都包含它。你现在对函数和类的工作原理已有大致的了解，可以开始使用其他程序员编写好的模块了。可以使用标准库中的任何函数和类，只需在程序开头包含一条简单的 import 语句即可。下面来看看模块 random，它在你模拟很多现实情况时很有用。

在这个模块中，一个有趣的函数是 randint() 。它将两个整数作为参数，并随机返回一个位于这两个整数之间（含）的整数。下面演示了如何生成一个位于 1 和 6 之间的随机整数：

```
>>> from random import randint
>>> randint(1, 6)
```

在模块 random 中，另一个有用的函数是 choice() 。它将一个列表或元组作为参数，并随机返回其中的一个元素：

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

第 8 章 练习

1. 创建一个 Die 类，它包含一个名为 sides 的属性，该属性的默认值为 6。编写一个名为 roll_die() 的方法，它打印位于 1 和骰子面数之间的随机数。创建一个 6 面的骰子再掷 10 次。

   创建一个 10 面的骰子和一个 20 面的骰子，再分别掷 10 次。
2. 创建一个列表或元组，其中包含 10 个数和 5 个字母。从这个列表或元组中随机选择 4 个数或字母，并打印一条消息，指出只要彩票上是这 4 个数或字母，就中大奖了。
3. 可以使用一个循环来明白前述彩票大奖有多难中奖。为此，创建一个名为 my_ticket 的列表或元组，再编写一个循环，不断地随机选择数或字母，直到中大奖为止。请打印一条消息，报告执行循环多少次才中了大奖。

第 9 章 Python 作业 1

利用 Python 实现学生管理系统该系统应具备的功能包括：

- 添加学生及成绩信息
- 将学生信息保存到文件中
- 修改和删除学生信息
- 查询学生信息
- 根据学生成绩进行排序
- 统计学生的总分