第8章 函数

在本章中，将学习编写函数（function/method）。 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。 通过使用函数，程序编写、阅读、测试和修复起来都更加容易。

还将学习向函数传递信息的方式； 学习如何编写主要任务是显示信息的函数，以及旨在处理数据并返回一个或一组值的函数； 最后，学习如何将函数存储在称为模块的独立文件中，让主程序文件的组织更为有序。

8.1 定义函数

下面是一个打印问候语的简单函数，名为greet_user() ：

In [11]:
```
def greet_user():
    """显示简单的问候语。"""
    print("Hello!")

greet_user()
```

```
Hello!
```

本例演示了最简单的函数结构。

使用关键字def来告诉Python，要定义一个函数。 这是函数定义，向Python指出了函数名，还可能在圆括号内指出函数为完成任务需要什么样的信息。 在这里，函数名为greet_user()，它不需要任何信息就能完成工作，因此括号是空的（即便如此，括号也必不可少）。 最后，定义以冒号结尾。 紧跟在def greet_user(): 后面的所有缩进行构成了函数体。 文本是称为文档字符串（docstring）的注释，描述了函数是做什么的。 文档字符串用三引号括起，Python使用它们来生成有关程序中函数的文档。 代码行print("Hello!") 是本函数体内的唯一一行代码，因此greet_user()只做一项工作：打印Hello! 。 要使用这个函数，可调用它。 函数调用让Python执行函数的代码。 要调用函数，可依次指定函数名以及用圆括号括起的必要信息。 由于这个函数不需要任何信息，调用它时只需输入greet_user()即可。和预期一样，它打印Hello!

8.1.1 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，如下所示：

In [12]:
```
def greet_user(username):
    """显示简单的问候语。"""
    print(f"Hello, {username.title()}!")

name = 'jesse'
greet_user(name)
```

```
Hello, Jesse!
```

8.1.2 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。

注意 大家有时候会形参、实参不分，因此如果你看到有人将函数定义中的变量称为实参或将函数调用中的变量称为形参，不要大惊小怪。

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

练习8-2： 喜欢的图书 编写一个名为favorite_book()的函数，其中包含一个名为title 的形参。 这个函数打印一条消息，下面是一个例子。 One of my favorite books is Alice in Wonderland.

8.2 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；
- 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。

8.2.1 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，如下所示：

In [13]:
```
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')
```

```
I have a hamster.
My hamster's name is Harry.
```

多次调用函数是一种效率极高的工作方式。 只需在函数中编写一次描述宠物的代码，然后每当需要描述新宠物时，都调用该函数并向它提供新宠物的信息。 即便描述宠物的代码增加到了10行，依然只需使用一行调用函数的代码，就可描述一个新宠物。 在函数中，可根据需要使用任意数量的位置实参，Python将按顺序将函数调用中的实参关联到函数定义中相应的形参。

位置实参的顺序很重要 使用位置实参来调用函数时，如果实参的顺序不正确，结果可能出乎意料：

In [14]:
```
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('harry', 'hamster')
```

```
I have a harry.
My harry's name is Hamster.
```

8.2.2 关键字实参

关键字实参 是传递给函数的名称值对。 因为直接在实参中将名称和值关联起来，所以向函数传递实参时不会混淆（不会得到名为Hamster的harry这样的结果）。 关键字实参让你无须考虑函数调用中的实参顺序，还清楚地指出了函数调用中各个值的用途。 下面来重新编写pets.py，在其中使用关键字实参来调用describe_pet()：

In [15]:
```
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(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry',animal_type='hamster')
```

```
I have a hamster.
My hamster's name is Harry.

I have a hamster.
My hamster's name is Harry.
```

8.2.3 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提供了实参时，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。

例如，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：

In [19]:
```
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('bob', 'cat')
#describe_pet('bob')
describe_pet(pet_name='willie')
```

```
I have a None.
My None's name is Willie.
```

请注意，在这个函数的定义中，修改了形参的排列顺序。 因为给animal_type 指定了默认值，无须通过实参来指定动物类型，所以在函数调用中只包含一个实参——宠物的名字。 然而，Python依然将这个实参视为位置实参，因此如果函数调用中只包含宠物的名字，这个实参将关联到函数定义中的第一个形参。 这就是需要将pet_name 放在形参列表开头的原因。

如果要描述的动物不是小狗，可使用类似于下面的函数调用：

In [22]:
```
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster','harry')
```

```
  Input In [22]
    describe_pet(animal_type='hamster','harry')
                                               ^
SyntaxError: positional argument follows keyword argument
```

8.2.4 等效的函数调用

鉴于可混合使用位置实参、关键字实参和默认值，通常有多种等效的函数调用方式。 请看下面对函数describe_pet()的定义，其中给一个形参提供了默认值：

```
def describe_pet(pet_name, animal_type='dog'):
```

基于这种定义，在任何情况下都必须给pet_name提供实参。 指定该实参时可采用位置方式，也可采用关键字方式。 如果要描述的动物不是小狗，还必须在函数调用中给animal_type提供实参。 同样，指定该实参时可以采用位置方式，也可采用关键字方式。

8.2.5 避免实参错误

等你开始使用函数后，如果遇到实参不匹配错误，不要大惊小怪。 你提供的实参多于或少于函数完成工作所需的信息时，将出现实参不匹配错误。 例如，如果调用函数describe_pet()时没有指定任何实参，结果将如何呢？

In [23]:
```
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()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [23], in <cell line: 6>()
      3     print(f"\nI have a {animal_type}.")
      4     print(f"My {animal_type}'s name is {pet_name.title()}.")
----> 6 describe_pet()

TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pe
t_name'
```

In [24]:
```
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('dog', 'Bob', 'male')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [24], in <cell line: 6>()
      3     print(f"\nI have a {animal_type}.")
      4     print(f"My {animal type}'s name is {pet_name.title()}.")
----> 6 describe_pet('dog', 'Bob', 'male')

TypeError: describe_pet() takes 2 positional arguments but 3 were given
```

8.3 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。

8.3.1 返回简单值

下面来看一个函数，它接受名和姓并返回整洁的姓名：

In [25]:
```
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)
```

```
Jimi Hendrix
```

8.3.2 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。 例如，假设要扩展函数get_formatted_name()，使其同时处理 中间名。为此，可将其修改成类似于下面这样：

In [ ]:
```
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提供了名和姓，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，并在用户没有提供中间名时不使用这个形参。 为让get_formatted_name() 在没有提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：

In [26]:
```
def get_formatted_name(first_name, last_name, middle_name=''):
    """返回整洁的姓名。"""

    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)
```

```
Jimi Hendrix
John Lee Hooker
```

8.3.3 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：

In [ ]:
```
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。 例如，下面的修改让你能存储年龄：

In [31]:
```
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)
musician = build_person('jimi', 'hendrix')
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix'}
```

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。

8.4 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。 下面使用函数来提高处理列表的效率。 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：

In [32]:
```
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', 'lisa', 'margret']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Lisa!
Hello, Margret!
```

我们将greet_users()定义为接受一个名字列表，并将其赋给形参names。 这个函数遍历收到的列表，并对其中的每位用户打印一条问候语。处定义了一个用户列表usernames，然后调用greet_users()并将该列表传递给它.

8.4.1 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：

In [ ]:
```
# 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：

In [34]:
```
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)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
[]
```

这个程序的输出与未使用函数的版本相同，但组织更为有序。 完成大部分工作的代码都移到了两个函数中，让主程序更容易理解。 只要看看主程序，就会发现这个程序的功能清晰得多.

描述性的函数名让别人阅读这些代码时也能明白，尽管没有任何注释。 相比于没有使用函数的版本，这个程序更容易扩展和维护。 如果以后需要打印其他设计，只需再次调用print_models()即可。 如果发现需要对打印代码进行修改，只需修改这些代码一次，就能影响所有调用该函数的地方。 与必须分别修改程序的多个地方相比，这种修改的效率更高。

该程序还演示了这样一种理念： 每个函数都应只负责一项具体的工作。 第一个函数打印每个设计，第二个显示打印好的模型。 这优于使用一个函数来完成这两项工作。 编写函数时，如果发现它执行的任务太多，请尝试将这些代码划分到两个函数中。 别忘了，总是可以在一个函数中调用另一个函数，这有助于将复杂的任务划分成一 系列步骤。

8.4.2 禁止函数修改列表

有时候，需要禁止函数修改列表。 例如，假设像前一个示例那样，你有一个未打印的设计列表，并编写了一个函数将这些设计移到打印好的模型列表中。

你可能会做出这样的决定： 即便打印好了所有设计，也要保留原来的未打印的设计列表，以供备案。 但由于你将所有的设计都移出了unprinted_designs，这个列表变成了空的，原来的列表没有了。 为解决这个问题，可向函数传递列表的副本而非原件。这样，函数所做的任何修改都只影响副本，而原件丝毫不受影响。

要将列表的副本传递给函数，可以像下面这样做：

```
function_name(list_name_[:])
```

切片表示法[:] 创建列表的副本。 如果不想清空未打印的设计列表，可像下面这样调用print_models()

```
print_models(unprinted_designs[:], completed_models)
```

这样函数print_models() 依然能够完成工作，因为它获得了所有未打印的设计的名称，但使用的是列表unprinted_designs的副本， 而不是列表unprinted_designs 本身。像以前一样，列表completed_models也将包含打印好的模型的名称，但函数所做的修改不会 影响到列表unprinted_designs。

虽然向函数传递列表的副本可保留原始列表的内容，但除非有充分的理由，否则还是应该将原始列表传递给函数。 这是因为让函数使用现成的列表可避免花时间和内存创建副本，从而提高效率，在处理大型列表时尤其如此。

8.5 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，但不管调用语句提供了多少实参，这个形参会将它们统统收入囊中：

In [37]:
```
def make_pizza(*toppings):
    """打印顾客点的所有配料。"""
    print(toppings)
    print(type(toppings))


make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
('pepperoni',)
<class 'tuple'>
```

```
('mushrooms', 'green peppers', 'extra cheese')
<class 'tuple'>
```

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。 注意，Python将实参封装到一个元组中，即便函数只收到一个值

In [38]:
```
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')
```

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

8.5.1 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：

In [39]:
```
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')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- 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。

注意 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。

8.5.2 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：

In [41]:
```
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)
```

```
{'location': 'princeton', 'field': 'physics', 'award': 'Nobel Prize', 'first_name': 'alb
ert', 'last_name': 'einstein'}
```

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。

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，不管额外提供多少个键值对，它都能正确地处理。

注意 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。

8.6 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。

In [ ]:
```
# goto 8-2_module_example
```

In [ ]:
```
github/bitbucket/subversion  -> hudson/Jinkins -> testing -> virtual
```