Chapter viii FUNCTIONS

8.1 Defining a Function

Here's a simple function named greet_user() :

```
def greet_user():
    """Display a simple greeting."""    #defning a function
    print("Hello!")

greet_user()                            #call a function
```

> `print (function.__doc__)` You can print the doc string of the function.

8.1.1 Passing Information to a Function

```
def greet_user(username):
    print("Hello, " + username.title() + "!")

greet_user('jesse')
```

```
Hello, Jesse!                                    #output
```

8.1.2 Arguments and Parameters

```
def greet_user(username):
    print("Hello, " + username.title() + "!")

greet_user('jesse')
```

The variable username in the definition of greet_user() is an example of a parameter.The value 'jesse' in greet_user('jesse') is an example of an argument.

Try It Yourself

8-1 Message: Write a function called display_message() that prints one sentence telling everyone what you are learning about in this chapter Call the function, and make sure the message displays correctly.

8-2 Favorite Book: Write a function called favorite_book() that accepts one parameter, title The function should print a message, such as One of my favorite books is Alice in Wonderland Call the function, making sure to include a book title as an argument in the function call

8.2 Passing Arguments

8.2.1 Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called positional arguments.

```
def describe_pet(animal_type, pet_name):
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is "+
                        pet_name.title() + ".")

describe_pet('hamster', 'harry')
```

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

1.Multiple Function Calls

```
describe_pet('hamster', 'harry')
describe_pet('dog', 'Willie')
```

2.Order Matters in Positional Arguments

```
describe_pet('harry', 'hamster')
```

8.2.2 Keyword Arguments

A keyword argument is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion (you won't end up138 Chapter 8 with a harry named Hamster). Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call。

The above program can be changed to the following case:

```
def describe_pet(animal_type, pet_name):
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " +
                        pet_name.title() + ".")

describe_pet(animal_type='hamster', pet_name='harry')
```

The order of keyword arguments doesn't matter because Python knows where each value should go. The following two function calls are equivalent:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

8.2.3 Default Values

When writing a function, you can define a default value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value.

```
def describe_pet(pet_name, animal_type='dog'):
    print("\nI have a " + animal_type + ".")
    print("My " + animal_type + "'s name is " +
                        pet_name.title() + ".")

describe_pet(pet_name='willie')
```

> Note that the order of the parameters in the function defnition had to be changed.
>
> When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values.

8.2.4 Equivalent Function Calls

Because positional arguments, keyword arguments, and default values can all be used together, often you'll have several equivalent ways to call a function.

All of the following calls would work for this function:

```
# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')
# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

8.2.5 Avoiding Argument Errors

Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.

try it:

```
describe_pet()
```

TRY IT YOUSELF

8-3 T-Shirt:Write a function called make_shirt() that accepts a size and the text of a message that should be printed on the shirt The function should print a sentence summarizing the size of the shirt and the message printed on it. Call the function once using positional arguments to make a shirt.Call the function a second time using keyword arguments.

8-4 . Large Shirts: Modify the make_shirt() function so that shirts are large by default with a message that reads. I love Python Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.

8-5 Cities: Write a function called describe_city() that accepts the name of a city and its country The function should print a simple sentence, such as Reykjavik is in Iceland Give the parameter for the country a default value Call your function for three different cities, at least one of which is not in the default country

8.3 Return Values

8.3.1 Returning a Simple Value

Let's look at a function that takes a first and last name, and returns a neatly formatted full name:

```
def get_formatted_name(first_name, last_name):
    full_name = first_name + ' ' + last_name
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

```
Jimi Hendrix
```

8.3.2 Making an Argument Optional

Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

For example, say we want to expand get_formatted_name() to handle middle names as well.But middle names aren't always needed.

```
def get_formatted_name(first_name, last_name, middle_name=''):
    if middle_name:
        full_name = first_name + ' ' + middle_name +
                                    ' ' + last_name
    else:
        full_name = 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)
```

This modified version of our function works for people with just a first and last name, and it works for people who have a middle name as well.

```
Jimi Hendrix
John Lee Hooker
```

8.3.3 Returning a Dictionary

A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries. For example, the following function takes in parts of a name and returns a dictionary representing

a person:

```
def build_person(first_name, last_name):
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

8.3.4 Using a Function with a while Loop

study independently

TRY IT YOURSELF

8-6 City Names: Write a function called city_country() that takes in the name of a city and its country The function should return a string formatted like this:

Album: Write a function called make_album() that builds a dictionary describing a music album .The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information .Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly. Add an optional parameter to make_album() that allows you to store the number of tracks on an album .If the calling line includes a value for the number of tracks, add that value to the album's dictionary. Make at least one new function call that includes the number of tracks on an album.

8-8 User Albums: Start with your program from Exercise 8-7 Write a while loop that allows users to enter an album's artist and title .Once you have that information, call make_album() with the user's input and print the dictionary that's created. Be sure to include a quit value in the while loop.

8.4 Passing a list

Say we have a list of users and want to print a greeting to each. The following example sends a list of names to a function called greet_users(), which greets each person in the list individually

```
def greet_users(names):
    for name in names:
       msg = "Hello, " + name.title() + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

8.4.1 Modifying a List in a Function

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work effciently even when you're dealing with large amounts of data.

Consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they're moved to a separate list. The following code does this without using functions:

```
 # Start with some designs that need to be printed.
unprinted_designs = ['iphone case', 'robot pendant',
                                    'dodecahedron']
completed_models = []
# Simulate printing each design, until none are left.
# Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()
   # Simulate creating a 3D print from the design.
    print("Printing model: " + current_design)
    completed_models.append(current_design)
# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

output:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: iphone case
The following models have been printed:
dodecahedron
robot pendant
iphone case
```

More efficient:

```
def print_models(unprinted_designs, completed_models):
        # Simulate printing each design, until none are left.
        # Move each design to completed_models after printing.
        while unprinted_designs:
                current_design = unprinted_designs.pop()
        #Simulate creating a 3D print from the design.
                print("Printing model: " + current_design)
        completed_models.append(current_design)
```

```
def show_completed_models(completed_models):
        # Show all the models that were printed.
        print("\nThe following models have been printed:")
        for completed_model in completed_models:
        print(completed_model)
```

Print:

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

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

8.4.2 Preventing a Function from Modifying a List

Sometimes you'll want to prevent a function from modifying a list. In this case, you can address this issue by passing the function a copy of the list, not the original.Any changes the function makes to the list will affect only the copy, leaving the original list intact.

You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

The slice notation [:] makes a copy of the list to send to the function.

TRY IT YOUSELF

8-9 Magicians: Make a list of magician's names Pass the list to a function called show_magicians(), which prints the name of each magician in the list.

8-10 Great Magicians: Start with a copy of your program from Exercise 8-9 .Write a function called make_great() that modifes the list of magicians by adding the phrase the Great to each magician's name Call show_magicians() to see that the list has actually been modifed.

8-11 Unchanged Magicians: Start with your work from Exercise 8-10 Call the function make_great() with a copy of the list of magicians' names .Because the original list will be unchanged, return the new list and store it in a separate list. Call show_magicians() with each list to show that you have one list of the original names and one list with the Great added to each magician's name.

8.5 Passing an arbitrary number of arguments

Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, *toppings, but this parameter collects as many arguments as the calling line provides::

```
def make_pizza(*toppings):
    print(toppings)

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

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

> The asterisk in the parameter name *toppings tells Python to make an empty tuple called toppings and pack whatever values it receives into this tuple. The print statement in the function body produces output showing that Python can handle a function call with one value and a call with three values.

8.5.1 Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function defnition.

For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter *toppings:

```
def make_pizza(size, *toppings):
    print("\nMaking a " + str(size) +
          "-inch pizza with the following toppings:")

for topping in toppings:
    print("- " + 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
```

8.5.2 Using Arbitrary Keyword Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function build_profile() in theFunctions 153 following example always takes in a frst and last name, but it accepts an arbitrary number of keyword arguments as well:

user_profile.py

```
def build_profile(first, last, **user_info):
    profile = {}
①   profile['first_name'] = first
    profile['last_name'] = last
②   for key, value in user_info.items():
        profile[key] = value
    return profile

user_profile = build_profile('albert', 'einstein',
       location='princeton', field='physics')
print(user_profile)
```

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

The definition of build_profile() expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter **user_info cause. Python to create an empty dictionary called user_info and pack whatever name-value pairs it receives into this dictionary. Within the function, you can access the name value pairs in user_info just as you would for any dictionary.

In the body of build_profile(), we make an empty dictionary called profile to hold the user's profile. At ❶ we add the first and last names to this dictionary ,because we'll always receive these two pieces of information from the user. At ❷ we loop through the additional key-value pairs in the dictionary user_info and add each pair to the profile dictionary. Finally, we return the profile dictionary to the function call line.

Summarize：

- Single asterisk function parameter.The parameters received by a single asterisk function parameter constitute a tuple.
- Double star function parameter.The parameters received by the double star function parameters constitute a dictionary.

TRY IT YOUSELF

8-12 Sandwiches: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that is being ordered .Call the function three times, using a different number of arguments each time.

8-13 User Profile: Start with a copy of user_profle.py from page 153 Build a profle of yourself by calling build_profile(), using your frst and last names and three other key-value pairs that describe you.

8-14Cars: Write a function that stores information about a car in a dictionary .The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature .Your function should work for a call like this one:

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

8.6 Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. By using descriptive names for your functions, your main program will be much easier to follow. You can go a step further by storing your functions in a separate file called a module and then importing that module into your main program. An import statement tells Python to make the code in a module available in the currently running program file.

8.6.1 Importing an Entire Module

To start importing functions, we first need to create a module. A module is a file ending in .py that contains the code you want to import into your program. Let's make a module that contains the function make_pizza(). To make this module, we'll remove everything from the file pizza. py except the function make_pizza():

pizza. py

```
def make_pizza(size, *toppings):
    print("\nMaking a " + str(size) +
       "-inch pizza with the following toppings:")
for topping in toppings:
    print("- " + topping)
```

Now we'll make a separate file called making_pizzas.py in the same directory as pizza. py. This file imports the module we just created and then makes two calls to make_pizza():

making_pizzas.py

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

When Python reads this file, the line import pizza tells Python toopen the file pizza.py and copy all the functions from it into this program.To call a function from an imported module, enter the name of the module you imported, pizza, followed by the name of the function, make_pizza(), separated by a dot.This code produces the same output as the original program that didn't import a module::

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

8.6.2 Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

The making_pizzas.py example would look like this if we want to import just the function we're going to use:

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

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function make_pizza() in the import statement, we can call it by name when we use the function

8.6.3 Using as to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique alias—an alternate name similar to a nickname for the function.

Here we give the function make_pizza() an alias, mp(), by importing make_pizza as mp. The as keyword renames a function using the alias you provide:

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

The import statement shown here renames the function make_pizza() to mp() in this program. Any time we want to call make_pizza() we can simply write mp() instead, and Python will run the code in make_pizza() while avoiding any confusion with another make_pizza() function you might have written in this program file.

The general syntax for providing an alias is:

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

8.6.4 Using as to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like p for pizza, allows you to call the module's functions more quickly. Calling p.make_pizza() is more concise than calling. pizza.make_pizza():

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

Calling the functions by writing p.make_pizza() is not only more concise than writing pizza.make_pizza(),but also redirects your attention from the module name and allows you to focus on the descriptive names of its functions. These function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

The general syntax for this approach is:

```
import module_name as mn
```

8.6.5 Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk (*) operator:

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

The asterisk in the import statement tells Python to copy every function from the module pizza into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get some unexpected results. Python may see several functions or variables with the same name, and instead of importing all the functions separately, it will overwrite the functions

The best approach is to import the function or functions you want,nor import the entire module and use the dot notation. This leads to clear code that's easy to read and understand.

8.7 Styling Functions

You need to keep a few details in mind when you're styling functions. Functions should have descriptive names, and these names should use lowercase letters and underscores. Descriptive names help you and others understand what your code is trying to do. Module names should use these conventions as well.

Every function should have a comment that explains concisely what the function does. This comment should appear immediately after the function defnition and use the docstring format. In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

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

The same convention should be used for keyword arguments in function calls:

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

.

Most editors automatically line up any additional lines of parameters to match the indentation you have established on the first line:

```
def function_name(
        parameter_0, parameter_1, parameter_2,
        parameter_3, parameter_4, parameter_5):
  function body...
```

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

All import statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

TRY IT YOURSELF

8-15 Printing Models: Put the functions for the example print_models.py in a separate file called printing_functions.py. Write an import statement at the top of print_models.py, and modify the file to use the imported functions.

8-16 Imports: Using a program you wrote that has one function in it, store that function in a separate file Import the function into your main program fle, and

call the function using each of these approaches:

```
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 *
```

Summary

One of your goals as a programmer should be to write simple code that does what you want it to, and functions help you do this. They allow you to write blocks of code and leave them alone once you know they work. When you know a function does its job correctly, you can trust that it will continue to work and move on to your next coding task.

Functions allow you to write code once and then reuse that code as many times as you want. When you need to run the code in a function, all you need to do is write a one-line call and the function does its job. When you need to modify a function's behavior, you only have to modify one block of code, and your change takes effect everywhere you've made a call to that function.

Using functions makes your programs easier to read, and good function names summarize what each part of a program does. Reading a series of function calls gives you a much quicker sense of what a program does than reading a long series of code blocks.

Functions also make your code easier to test and debug. When the bulk of your program's work is done by a set of functions, each of which has a specifc job, it's much easier to test and maintain the code you've written. You can write a separate program that calls each function and tests whether each function works in all the situations it may encounter. When you do this, you can be confdent that your functions will work properly each time you call them.