# FUNCTIONS


Functions allow you to define a set of actions and then repeat that set of actions at any time by calling the function name in your code. When creating a function, you can specify exactly the information it needs to do its work, and your function can return exactly the values you need it to.

Using functions will help you write code that is well organized, concise, and easy to build upon.

# ABOUT FUNCTIONS

- What is a function?
- How do you define functions?
- How do you call functions?

**A *function* is a block of code you give a name to so you can use it again. Calling its name runs the entire block of code.**

This one-line function displays a greeting:

```
def hello():
    """Greet everyone."""
    print("Hello, everyone!")

hello()
```

To create a function, *define* it with the keyword `def`, followed by the function's name, a set of parentheses, and a colon at the end.

The next line is a *docstring*: a short description of the function in triple quotes. The function's *body* contains the Python code that runs when you call the function. The body has no line limit and is indented under the function definition.

To *call* a function, use the function's name followed by a set of parentheses. Here's the output from calling our `hello()` function:

```
Hello, everyone!
```

# PASSING ARGUMENTS

- What is a parameter?
- What is an argument?
- How do you write a function that takes in information?
- How do you pass arguments to functions?

**A *parameter* is information the function needs to do its work. An *argument* is a value passed to a function.**

To define parameters for a function, place them within the parentheses in the function's definition. This creates a `hello()` function that requires `name` data:

```
def hello(name):
    """Greet someone by name."""
    print(f"Hello, {name}!")
```

To run, this function needs a name argument, which it stores in the `name` variable and uses when it calls `print()`.

Pass an argument to a function by including a value in the parentheses when you call the function:

```
hello("Gretchen")
```

The argument provides a value for the parameter `name`, allowing the `hello()` function to generate a personalized greeting:

```
Hello, Gretchen!
```

Sometimes *parameter* and *argument* are used interchangeably.

# POSITIONAL ARGUMENTS

- What are positional arguments?
- How do you use positional arguments in a function call?
- What happens if you use positional arguments in the wrong order?

**When a function needs more than one value, it must match the values it receives to its parameters. Using *positional* arguments, the function matches the first value it receives to the first parameter, and so on.**

This function accepts a message and a username and posts a message summary:

```
def post_msg(msg, user):
    """Post a message from a user."""
    print(f'{user}: {message}')

post_msg("I love the internet!", "Willie")
post_msg("Me too!", "Ever")
```

```
Willie: I love the internet!
Ever: Me too!
```

The order of the arguments must match the order of the parameters in the function definition, or the output won't make sense:

```
post_msg("Willie", "I love the internet!")
```

This swaps the username and message values:

```
I love the internet!: Willie
```

# ARBITRARY POSITIONAL ARGUMENTS

- How do you write a function that accepts an unknown number of arguments?
- How do you call a function that accepts an arbitrary number of arguments?

**An asterisk (*) before a parameter name allows a function to receive any number of positional arguments. Use it when you don't know how many arguments the function will receive.**

This function accepts a username and the number of messages the user posts. The arguments make up a tuple:

```
def post_messages(user, *msgs):
    """Post all of a user's messages."""
    print(f"{user} said:")
    for msg in msgs:
        print(f"  {msg}")

post_messages('Ahna', "I love mountains.",
        "I love rivers.",
        "I love the ocean.")
```

The first argument must provide a value for the first parameter. The rest are placed into the tuple `msgs`. Here's the output:

```
Ahna said:
  I love mountains.
  I love rivers.
  I love the ocean.
```

A function can only have one parameter that collects an arbitrary number of positional arguments.

# KEYWORD ARGUMENTS

- What are keyword arguments?
- How do you use keyword arguments in a function call?

***Keyword arguments* specify the parameter a value will be assigned to so you can pass arguments in any order.**

In the function call, include the name of the parameter and the value to assign to that parameter:

```
def post_msg(msg, user):
    """Post a message from a user."""
    print(f'{user}: {message}')

post_msg(user="Willie",
         msg="I love the internet!")
```

Python assigns the value `"Willie"` to the parameter `user` and the value `"I love the internet!"` to the parameter `msg`:

```
Willie: I love the internet!
```

The order of keyword arguments in a function call doesn't matter.

# ARBITRARY KEYWORD ARGUMENTS

- How do you write a function that accepts an arbitrary number of keyword arguments?
- How do you call a function with an arbitrary number of keyword arguments?

**Placing a double asterisk (**) before a parameter allows a function to accept an arbitrary number of keyword arguments. Use them when you don't know the kind of information the function will receive.**

A parameter with ** collects any remaining name-value pairs from the function call and adds them to a dictionary. This function accepts an arbitrary amount of user information:

```
def desc_user(user, **desc):
    """Describe a user."""
    print(f"Description of {user}:")
    for key, value in desc.items():
        print(f"  {key}: {value}")

desc_user('anton', active=True,
        email='anton@example.com',
        num_posts=578)
```

The first argument must be the user. Then any remaining keyword arguments are packed into the `desc` dictionary:

```
Description of anton:
  active: True
  email: anton@example.com
  num_posts: 578
```

Each function can only have one parameter that collects an arbitrary number of keyword arguments.

# DEFAULT VALUES

- How do you assign a default value for a parameter?
- How do you call a function that has a default value for a parameter?

**Setting a parameter's *default value* allows the function call to use the default value unless another value is specified with the call.**

This function serves a cup of coffee. You can specify the cup size, but if you don't, you'll get a 12 oz cup:

```
def coffee(size=12):
    """Serve a cup of coffee."""
    print(f"Here's your {size} oz coffee!")

coffee()
coffee(16)
```

Here we get a 12 oz cup and a 16 oz cup:

```
Here's your 12 oz coffee!
Here's your 16 oz coffee!
```

Default values are helpful because they make some arguments optional.

# RETURN VALUES

- What is a return value?
- How do you write a function that returns a value?

**A function sends a *return value* back to the line that called the function, so the value can be used later in the code. The calling line uses a variable to store the return value.**

This function accepts a rectangle's width and height and returns the area:

```
def area_rect(w, h):
    """Calculate the area of a rectangle."""
    area = w * h
    return area

my_area = area_rect(5, 4)
print(my_area)
```

When the function is called, the return value is stored in the variable `my_area`. Here's the output:

```
20
```

You can return any kind of value. To return more than one value, place the values in parentheses to return them as a tuple.

# MODULES

- What is a module?
- How do you create a module?

**A *module* is a Python file that contains multiple functions or classes. By storing functions in this separate file, you can call them in programs without having to enter the full function code each time.**

This *area_functions.py* module has two functions:

```
"""A set of functions for calculating areas."""

def area_rect(w, h):
    """Calculate the area of a rectangle."""
    --snip--

def area_circle(r):
    """Calculate the area of a circle."""
    --snip--
```

No function *calls* are in this file, so this code doesn't run until the main code calls it. When you add to and improve functions in modules, the changes are immediately available to all programs that use them.

# IMPORTING FUNCTIONS

- How do you import an entire module and use each of the functions in the module?
- How do you give a module an alias when importing it?

**Use the `import` keyword to import a module and access its functions.**

After importing, use any module function by providing the module's name, a dot, and the function's name:

```
import area_functions

my_area = area_functions.area_rect(4, 5)
print(my_area)
```

This code has the same output as if it contained all the function code:

```
20
```

Use `as` to give a module an alias and use shorter function calls. Here we use `af` for `area_functions`:

```
import area_functions as af

my_area = af.area_rect(4, 5)
print(my_area)
```

# IMPORTING SPECIFIC FUNCTIONS

- How do you import one function from a module?
- How do you import several individual functions from a module?

**Use `from` with `import` to import a specific function from a module rather than the entire module. Then use the function's name as if it were defined within the program.**

Here we import from the `area_functions` module from Card 6.9:

```
from area_functions import area_rect

my_area = area_rect(4, 5)
print(my_area)
```

To import multiple functions from one module, separate them with commas:

```
from area_functions import area_rect, area_circle
```

Use an asterisk to import all the functions in a module at once:

```
from area_functions import *
```

This last approach is not recommended because it can cause conflicts. A module function might override a program function if they have the same name. If you need all the module functions, import the entire module and use an alias.