Overview

In this lesson, you will learn how to create functions that take input values called arguments. You will learn the difference between parameters and arguments, how to write functions that accept them, how to allow a function to accept any number of arguments, how to use type hints, and how to indicate when a parameter may not be used.

Important Information

Functions become much more powerful when they can receive data from outside the function. This data is passed into the function when it is called.

Parameters vs Arguments

Although they are closely related, **parameters** and **arguments** are not the same thing.

- **Parameters** are the variable names listed in a function definition.
- **Arguments** are the actual values passed into the function when it is called.

Example:

```
def greet(name):
    print(name)

greet("Alex")
```

- `name` is a **parameter**
- `"Alex"` is an **argument**

Functions With Parameters

Parameters act like local variables inside the function. Their values come from the arguments provided when the function is called.

```
def show_number(number):
    print(number)

show_number(5)
show_number(42)
```

Each time the function is called, the parameter `number` receives a different value.

Functions With Multiple Parameters

Functions can take more than one parameter.

```
def introduce(name, age):
    print(name)
    print(age)

introduce("Sam", 14)
```

The order of arguments matters. The first argument goes to the first parameter, the second argument goes to the second parameter, and so on.

Any Number of Arguments

Sometimes you do not know how many arguments will be passed into a function. Python allows this using `*args`.

- `*args` collects all extra arguments into a tuple
- The name `args` is a convention, but the `*` is what matters

```
def show_all(*args):
    print(args)

show_all(1, 2, 3)
show_all("a", "b", "c", "d")
```

This allows the function to accept any number of arguments without changing the function definition.

Type Hinting Arguments

Type hints allow you to indicate what type of data a parameter is expected to receive. This does not stop the program from running, but it helps with readability and error checking.

```
def repeat_word(word: str, times: int):
    print(word)
    print(times)
```

Type hints make code easier to understand and help code editors warn you about possible mistakes. However, in python, type hints **do not** change how the program runs. They really are just hints, and not hard and fast rules.

Unused Parameters

Sometimes a function must accept a parameter even if it is not used yet. In this case, the parameter name should start with an underscore `_`.

This signals to other programmers (and tools) that the parameter is intentionally unused.

```
def placeholder(_unused_value):
    print("This function ignores its parameter")
```

Using `_` helps keep code clean and avoids confusion.

Set Up

Create a new python file called `function_arguments.py`.

Copy, Change, Challenge - Using Parameters

Copy

```
def greet(name):
    print("Hello")
    print(name)

greet("Jordan")
```

Change

Change the argument so a different name is printed.

Challenge

Call the function three times using three different names.

Copy, Change, Challenge - Any Number of Arguments

Copy

```
def add_numbers(*numbers):
    print(numbers)

add_numbers(1, 2)
add_numbers(3, 4, 5, 6)
```

Change

Call the function with only one argument, then with at least five arguments.

Challenge

Modify the function so it prints each value in `numbers` on its own line.

Copy, Change, Challenge - Unused Parameters

Copy

```
def display_info(name: str, _id_number: int):
    print(name)

display_info("Taylor", 12345)
```

Change

Rename the `_id_number` parameter to something else that still starts with `_`.

Challenge

Create a function with two parameters where one is used and the other is intentionally unused, and demonstrate calling the function.