Functions

Functions in Python are reusable blocks of code that can be executed whenever you need to perform a specific task. They can take inputs and produce outputs, and are a powerful tool for organizing and simplifying your code. To define a function in Python, you use the `def` keyword followed by the name of the function, any parameters it takes (in parentheses), and a colon. The body of the function is indented below the definition. The syntax for defining a function is as follows:

```python
def function_name(input1, input2, ...):
    """Optional docstring describing function."""
    # do something
    return output # optional
```

Defining and calling a function

Here's an example of defining and calling a function:

```python
# Define the function
def greet():
    pass

greet()
```

In this example, we define a function called `greet`. It currently takes no arguments and uses the pass statement as a placeholder to do nothing. We will fill out this function in the next section.

Parameters and Arguments

When you define a function, you can specify one or more parameters that the function expects to receive when it is called. Parameters are like placeholders for values that will be passed in as arguments when the function is called.

Here's an example of a function with one parameter:

```python
def greet(name):
    print("Hello, " + name + "!")
```

In this case, `name` is the parameter, and it is used inside the function to print a personalized greeting.

When you call a function, you pass in values for each of its parameters, which are called arguments. Here's an example of calling the `greet` function with an argument:

```
greet("Alice")
```

In this case, we pass in the string `"Alice"` as the argument for the name parameter of the `greet` function.

Return Values

Functions in Python can also return values, which can be stored in variables or used in other parts of your code. To return a value from a function, you use the `return` keyword followed by the value you want to return.

Here's an example of a function that returns a value:

```
def add_numbers(a, b):
    return a + b
```

When you call this function, it returns the sum of the two numbers you pass in as arguments:

```
result = add_numbers(2, 3)
print(result)
```

In this case, we call the `add_numbers` function with the arguments `2` and `3`, and it returns the value `5`, which we store in the `result` variable and print to the console.

Returning Multiple Values

A function can return multiple values by using a tuple or a list. For example, the following function returns two values:

```
def get_values():
    return 1, 2

x, y = get_values()
print(x)
print(y)

z = get_values()
print(z)
```

Function Arguments and Keyword Arguments

In Python, we have two types of function arguments:

1. Positional arguments: These are the arguments that are passed to a function in a specific order.
2. Keyword arguments: These are the arguments that are passed to a function using the keyword argument syntax, which allows us to specify the argument name and its value.

Let's see an example of both types of arguments being passed to a function call:

```
def func(a, b, c):
    print("a:", a)
    print("b:", b)
    print("c:", c)
```

```
func(1, 2, 3)
```

```
func(b=2, c=3, a=1)
```

In the above example, the function `func` takes three arguments, `a`, `b`, and `c`.

In the first call, we passed three arguments, which are all positional arguments. The first argument is assigned to `a`, the second to `b`, and the third to `c`. In the second call, we passed the arguments using keyword arguments, so the order does not matter.

Default Parameter Values

You can also provide default values for function parameters, which will be used if no argument is passed in for that parameter. Here's an example:

```
def greet(name="world"):
    print("Hello, " + name + "!")
```

In this case, we define a `greet` function with a default parameter value of `"world"`. If you call the function with no arguments, it will use the default value:

```
greet()
```

But you can also override the default value by passing in a different argument:

```
greet("Alice")
```

In this case, we pass in the string `"Alice"` as the argument for the name parameter of the greet function, and it overrides the default value.

Variable Scope

Local Variables

Variables that are defined inside a function are said to have local scope. This means that they can only be accessed from within the function, and not from outside of it.

Here's an example:

```
def print_number():
    number = 42
    print(number)

print_number()
```

```
print(number)
```

In this case, the `number` variable is defined inside the `print_number` function, and can only be accessed from within that function. When we try to access it from outside the function, we get a `NameError`.

Global Variables

If you want to use a variable inside a function and also access it outside of the function, you can define it as a global variable. To do this, you use the global keyword followed by the name of the variable, inside the function.

Here's an example:

```
number = 42

def print_number():
    global number
    print(number)

print_number()
```

```
print(number)
```

Lambda Functions

Lambda functions are small, anonymous functions in Python that can be defined inline and used wherever a function is required. They are also known as "anonymous functions", because they do not have a name like regular functions do.

The syntax of a lambda function is very simple. It consists of the keyword lambda, followed by one or more arguments separated by commas, followed by a colon, and then the expression to be returned by the function. Here's the general syntax of a lambda function:

```
lambda arguments: expression
```

Here's an example of a lambda function that takes a single argument and returns its square:

```
lambda x: x ** 2
```

Here's an example of a lambda function that takes two arguments and returns their sum:

```
lambda x, y: x + y
```

Lambda functions are useful when you need to define a simple function that will only be used once, and you don't want to define a named function for it.

Using Lambda Functions

Lambda functions are commonly used with other built-in Python functions, such as `map()`, `filter()`, and `reduce()`. These functions take a function as their first argument, and then apply that function to a sequence of values.

Here's an example of using a lambda function with the `map()` function:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

Finally, lambda functions can be assigned to variables, which allows you to use them like regular functions. For example:

```python
square = lambda x: x ** 2
print(square(5))
```

Writing Functions With an Arbitrary Number of Arguments

`*args` and `**kwargs` are special syntax in Python that allow you to pass a variable number of arguments to a function. The term "args" stands for "arguments", and "kwargs" stands for "keyword arguments".

`*args` allows you to pass any number of positional arguments to a function, which are then collected into a tuple. This can be useful when you don't know how many arguments a function will need to take ahead of time. Here's an example:

```python
def my_function(*args):
    for arg in args:
        print(arg)

my_function(1, 2, 3)
```

`**kwargs` allows you to pass any number of keyword arguments to a function, which are then collected into a dictionary. This can be useful when you want to pass in a large number of arguments, and you want to be able to refer to them by name inside the function. Here's an example:

```python
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

my_function(name="Alice", age=30, location="London")
```

Using `*args` and `**kwargs` Together

You can also use `*args` and `**kwargs` together in the same function definition. In this case, the positional arguments will be collected into the `args` tuple, and the keyword arguments will be collected into the `kwargs` dictionary. Here's an example:

```python
def my_function(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(key, value)

my_function(1, 2, 3, name="Alice", age=30, location="London")
```

Docstrings

It is good practice to include a docstring for each function you define. A docstring is a string that describes what the function does, and is enclosed in triple quotes. Like comments in Python, docstrings are ignored by the Python interpreter, but they are used by other tools, such as IDEs and documentation generators. Here's an example:

```python
def print_number(number):
    """Prints the input number."""
    print(number)
```

Exercises

1. Write a function that takes two numbers as arguments and returns their sum.
2. Write a function that takes a string as an argument and returns the string reversed.
3. Write a function that takes a list of numbers as an argument and returns the largest number in the list.
4. Write a function that takes a list of strings as an argument and returns the longest string in the list.
5. Write a function that takes a string as an argument and returns a count of how many vowels are in the string.
6. Write a function that takes a list of numbers as an argument and returns a new list with each number squared.
7. Write a function that takes a list of strings as an argument and returns a new list with each string capitalized.
8. Write a function that takes a list of numbers as an argument and returns the average of the numbers.
9. Write a function that takes two strings as arguments and returns `True` if they are anagrams (contain the same letters, regardless of order), and `False` otherwise.
10. Write a function that takes a list of numbers as an argument and returns the median of the numbers (the middle value, if the list is sorted).

11. Write a lambda function that takes a list of integers and returns a new list with each integer multiplied by 2.
12. Write a lambda function that takes a list of strings and returns a new list with each string in uppercase.
13. Write a function that takes any number of arguments using *args and returns the average of all the arguments.
14. Write a function that takes any number of keyword arguments using **kwargs and returns a new dictionary with all the key-value pairs reversed, i.e. where the keys are the values and the values are the keys.
15. Below is a complex Python script that performs several tasks related to processing a list of students and their grades. Rewrite the script using functions to make it more readable and maintainable. This is a process called refactoring, and it is an important part of writing good code.

```python
students = [
{"name": "Alice", "grades": [90, 85, 88]},
{"name": "Bob", "grades": [80, 75, 78]},
{"name": "Charlie", "grades": [87, 91, 94]},
{"name": "David", "grades": [76, 82, 85]},
{"name": "Eve", "grades": [90, 92, 95]}
]

# Calculate the average grade for each student
for student in students:
    total = 0
    count = 0
    for grade in student["grades"]:
        total += grade
        count += 1
    average = total / count
    student["average"] = average

# Find the highest average
highest_average = 0
top_student = None
for student in students:
    if student["average"] > highest_average:
        highest_average = student["average"]
        top_student = student["name"]
```

```
# Print the top student and their average
print(f"The top student is {top_student} with an average grade of {highest_average:.2

# Find the lowest average
lowest_average = 100
bottom_student = None
for student in students:
    if student["average"] < lowest_average:
        lowest_average = student["average"]
        bottom_student = student["name"]

# Print the bottom student and their average
print(f"The bottom student is {bottom_student} with an average grade of {lowest_avera

# Calculate the overall class average
total_class_average = 0
for student in students:
    total_class_average += student["average"]
total_class_average /= len(students)

# Print the class average
print(f"The class average is {total_class_average:.2f}")
```

The script performs the following tasks:

```
1. Calculate the average grade for each student.
2. Find the student with the highest average grade.
3. Find the student with the lowest average grade.
4. Calculate the overall class average.
```

These tasks can be refactored into separate functions to improve code readability, maintainability, and testability.

Project: Temperature Converter

Create a Python program that converts temperature from Fahrenheit to Celsius and vice versa. The program should use functions to perform the conversions.

Project Requirements

1. The program should allow the user to convert temperature from Fahrenheit to Celsius and vice versa.

2. The program should use functions to perform the conversions.
3. The program should handle errors appropriately, for example, when the user enters non-numeric inputs.

Example Output

```
Temperature Converter

Choose an option:
1. Fahrenheit to Celsius
2. Celsius to Fahrenheit

Enter your choice (1/2): 1

Enter temperature in Fahrenheit: 32
32°F is 0°C.
```

Bonus Projects: Converting to Functions

Try to refactor your previous project code to use functions! For example, your `Simple Calculator` project could be adapted to use functions for each operation. You can also try to refactor your `Grocery List` project to use functions. Refactoring code is a common practice in programming, and it is a great way to improve your code. Having reproducible code in functions will make it easier to reuse, test and debug your code, as well as reducing the chances of errors creeping in in the first place! It will also make your code more readable and easier to maintain, which is important for larger projects and projects with multiple collaborators.

Further Reading

Check out these resources to learn more about functions in Python:

- W3Schools
  - Python Functions
- Real Python
  - Python Functions
  - Python Function Arguments
  - Python Docstrings
- Python Documentation