AI Boot Camp

Programming with Functions — Part 1

Module 3 Day 1

Class Objectives

By the end of class, you will be able to:

1. Understand the importance of functions.
2. Create, write, and use Python functions.
3. Perform transformations on iterable objects using Python built-in functions.
4. Document and style code using the latest standards.

Instructor **Demonstration**

Why Use Functions?

Functions

<DRY>

Don’t Repeat Yourself

In software engineering, Don’t Repeat Yourself (DRY) is a principle of software development that we can use functions and modules to avoid repeating code.

Functions

A function is a block of reusable code that can be used to perform an action.

A function provides better:

Organization

- Functions make code more readable because blocks of code are condensed (or *wrapped*) into a single, callable name.

Modularity

- Functions can be called multiple times and used over and over again.

Comprehension

- In long, complex tasks, code can be split up into smaller blocks, resulting in a more readable code.
- Function names are often good indicators of what the function will be trying to achieve.

Instructor **Demonstration**

Getting Started with Functions

Function **Anatomy**

Recall that you were introduced to list functions `len()`, `max()`, `min()`, `sum()`, `append()`, `pop()`, and `remove()`.

For each of the following functions, think about:

- Does the name of the function align with the task the function does?
- What do you expect the function to do (or **return**)?

The Anatomy of a Function

A: The function definition

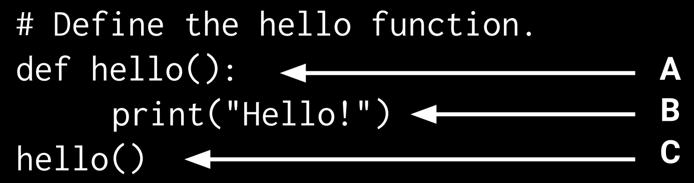
- Defines the function using the **`def`** keyword.
- The word **`hello`** is the function name, followed by parenthesis and a colon.

B: The body of the function

- Contains the code that the function will execute.
- Must be indented (4 spaces) to avoid errors.

C: The function call

- Tells the Python interpreter to run the code inside the function.



```
# Define the hello function.
def hello():
    print("Hello!")
hello()
```

Function recap

Consider this `scream` function example. Which part of the following is calling the function?

```
def scream():
    print(“AAAH!”)
scream()
scream()
scream()
print("Okay, I feel much better.")
```

The output is:

```
AAAAH!
AAAAH!
AAAAH!
Okay, I feel much better.
```

The `name==main` idiom

In Python programming, it is common to use the conditional statement, or idiom, `if __name__ == "__main__"` to house the code that you actually want to execute.

```
# Define the scream function
def scream():
    print("AAAAH!")

# Name-main idiom with the function call.
if __name__ == "__main__":
    scream()
    scream()
    scream()
    print("Okay, I feel much better.")
```

The output is:

```
AAAAH!
AAAAH!
AAAAH!
Okay, I feel much better.
```

Activity:

A Definitive Buy

In this activity, you'll create a function that will be used in the back end of a new payment processing system. The aim of the task is to write a single, reusable function that will be called each time a payment occurs in the system.

Suggested Time:

10 Minutes

Time’s up!

Let’s review

Questions?

Instructor **Demonstration**

Function Variables and Parameters

Function Scopes: Local and Global Variables

Python makes use of the concept of **scope** to prevent unintended mistakes caused by naming collisions. **Scope** refers to the part of the program that has "knowledge" of a variable.

There are two common types of scopes:

01

Local Variables

- A local variable is only accessible to the body of the function.
- You can't access the local variable with code outside the function.

02

Global Variables

- A global variable is accessible to any code outside of the function.

Local Variables

Local variables are only accessible to the body of the function. If you try accessing the local variable outside the function you'll get back an error.

```
def add():
    """This function takes two numbers and adds them and then returns the total."""
    first_number = 1 # This is a local scope of the function.
    second_number = 2 # This is a local scope of the function.
    total = first_number + second_number
    print(f"Your total is: {total}")

if __name__ == "__main__":
    add()
    print(first_number)
```

The output is:

```
Your total is: 3
Traceback (most recent call last):
line 14, in <module>
    print(first_number)
NameError: name first_number is not defined
```

Global Variables

A global variable is accessible to any code outside of the function.

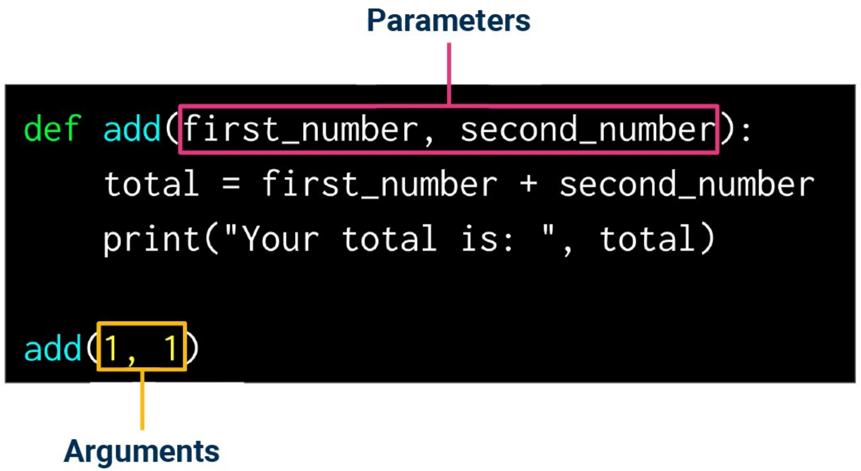
```
# Global variables for first_number and second_number.
first_number = 2
second_number = 3

# Define a function that will add two numbers.
def add():
    first_number = 4 # This is a local variable of the function.
    second_number = 6 # This is a local variable of the function.
    total = first_number + second_number
    print("Your total is: ", total)

if __name__ == "__main__":
    add()
    print(f"The global variables for the 'first_number' and 'second_number` are {first_number} and
{second_number}")
```

The output is:

```
Your total is: 10
The global variables for the first_number and second_number are 2 and 3.
```

Python Parameters and Arguments

1. Parameters are variables used in a function that refer to the data. The function uses the data to perform its actions.
2. The function gets the data in the form of arguments.
3. Arguments can be numbers, strings, predefined variables, lists, dictionaries, and tuples.



```python
def add(first_number, second_number):
    total = first_number + second_number
    print("Your total is: ", total)

add(1, 1)
```

Python Parameters and Arguments

There are five common ways of passing arguments to functions:

1. Positional
2. Keyword
3. Combination of positional and keyword
4. Iterable unpacking
5. Dictionary unpacking

Positional Arguments

Positional arguments are assigned to the parameter in the corresponding “position” within the function.

```python
def savings(balance, apr, days):
    """This function adds three positional arguments,
    adds them and prints the total"""
    interest_rate = (apr/100) * (days/365)
    interest_earned = balance * interest_rate
    balance += interest_earned
    print(f"The new balance is: {balance}")

if __name__ == "__main__":
    # Incorrect positional order
    savings(5, 31, 50000)
    # Correct positional order
    savings(50000, 3, 31)
```

Keyword Arguments

1 Keyword arguments are assigned to parameters using the `name=value` syntax.

2 Unlike with positional arguments, the order of the arguments doesn't matter with keyword arguments which makes the code more readable and easier to debug.

```
def keyword(a, b, c):
    """This function takes three keyword arguments,
    adds them, and prints the total"""
    total = a + b + c
    print(f"The total is: {total}")

if __name__ == "__main__":
    keyword(a=-3, c=10, b=5)
```

Positional and Keyword Arguments

You can use a combination of positional and keyword arguments at the same time, but the positional arguments have to be listed before the keyword arguments.

```
def pos_key_args(a, b, c):
    """This function takes one positional argument
    and two keyword arguments, adds them, and prints the total"""
    total = a + b + c
    print(f"The total is: {total}")

if __name__ == "__main__":
    pos_key_args(42, b=-10, c=5)
```

Iterable Unpacking

1. Iterables, like list and tuples, can be passed as parameters.
2. When we pass an iterable to a function we “unpack” the iterable using the `*iterable_name` syntax.
3. Each item in the iterable is passed as a positional argument to the function.

```
def iterable(a, b, c):
    """This function takes an iterable
    (list of tuple), adds the values in
    the iterable, and prints the total"""
    total = a + b + c
    print(f"The total is: {total}")

if __name__ == "__main__":
    tuple_values = (5, -10, 7)
    list_values = [7, 23, -11]
    iterable(*tuple_values)
    iterable(*list_values)
```

Dictionary Unpacking

1. Python dictionaries can be passed as parameters.
2. When we pass an iterable to a function we “unpack” the values of the dictionary using the `**dictionary_name` syntax.
3. If you use the `*dictionary_name` syntax you will pass the keys to the function.

```
def dictionary(a, b, c):
    """This function takes an iterable (list
    Or tuple), adds the values in the iterable,
    and prints the total"""
    total = a + b + c
    print(f"The total is: {total}")

if __name__ == "__main__":
    dict_values = {'b': -4, 'c': 100, 'a':-42 }
    dictionary(*dict_values) # returns the keys
    dictionary(**dict_values) # returns the values
```

Activity:

Car Loan Calculator

In this activity, you'll create a function that calculates the future value of a car loan based on the current loan value, months remaining for the loan, and the annual interest rate.

Suggested Time:

10 Minutes

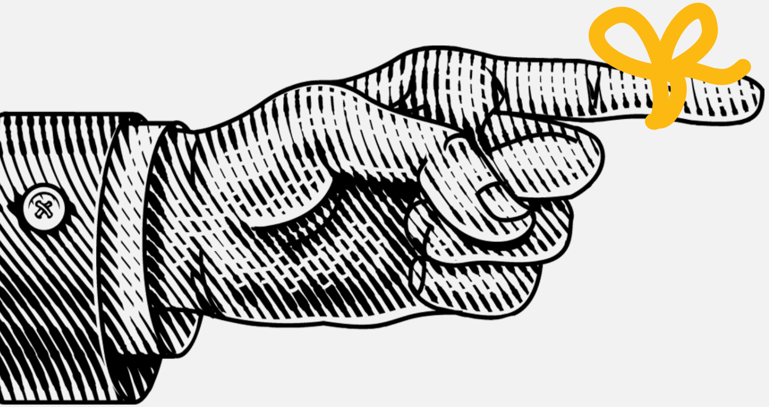
Time's up!

Let's review

Questions?

Instructor **Demonstration**

Returning Values



Reminder: Functions

A function is a block of reusable code that can be used to perform an action. Functions are used to improve code's **organization**, **modularity**, and **comprehension**. You have defined functions and variables, and set function parameters in the form of arguments.

Now, consider the following function:

```
def average_numbers(numbers):
    """ Calculates the average of an array of numbers"""
    average = sum(numbers) / len(numbers)
    print("The average is: ", average)

if __name__ == "__main__":
    average_numbers([1, 2, 3])
```

The output is:

```
The average is 2.0
```

Returning values

1. The function **`return`** statement allows the results of the function to be returned to the function call.

2. The **`return`** statement allows us implement code modularity—one of the benefits of functions.

```python
def average_numbers(numbers):
    """ Calculates the average of an array of numbers"""
    average = sum(numbers) / len(numbers)
    return average

if __name__ == "__main__":
    first_average = average_numbers([1, 2, 3])
    second_average = average_numbers([4, 5, 6])
    print(f'The average of the first list is {first_average}')
    print(f'The average of the second list is {second_average}')
```

The output is:

```
The average of the first list is 2.0
The average of the second list is 5.0
```

Activity:

Returned Goods

In this activity, you'll create a function that automatically determines the firm's weekly total insurance payouts each based on the number of weekly claims.

Suggested Time:

10 Minutes

Time’s up!

Let’s review

Break

15 mins

Instructor **Demonstration**

Anonymous and Built-In Functions

Anonymous and built-in functions

There are many built-in Python functions that make programming much easier. Three of these functions are:

1. `map`
2. `lambda`
3. `filter`

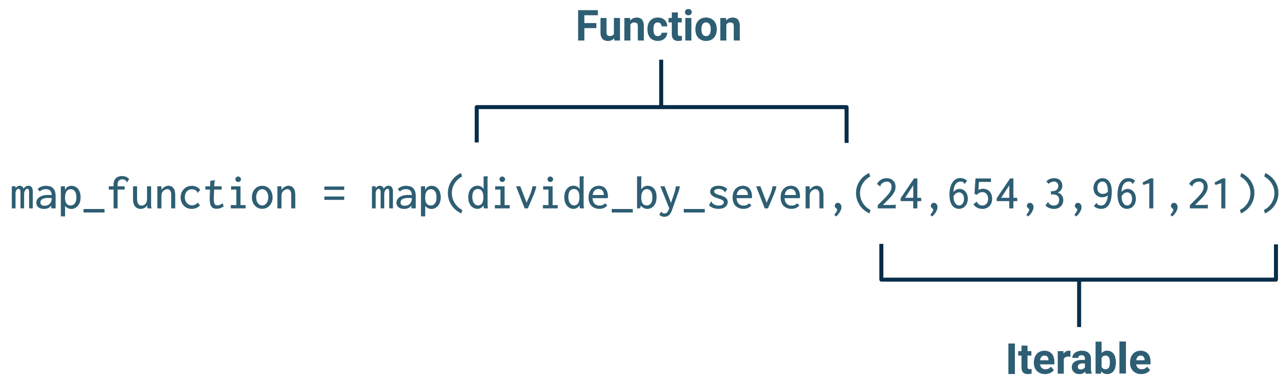
Understanding these functions will allow us to work with iterable objects like lists without using cumbersome `for` loops.

The `map` function

The `map` function is similar to functions that use list comprehension because its purpose is to apply a function to each iteration in an iterable object.

When you call the `map` function, you must provide two arguments. Your function call will look like this: `map(function, iterable)`.

- The `function` object is a function that we want to apply.
- The iterable object is what we want to apply `function` across.

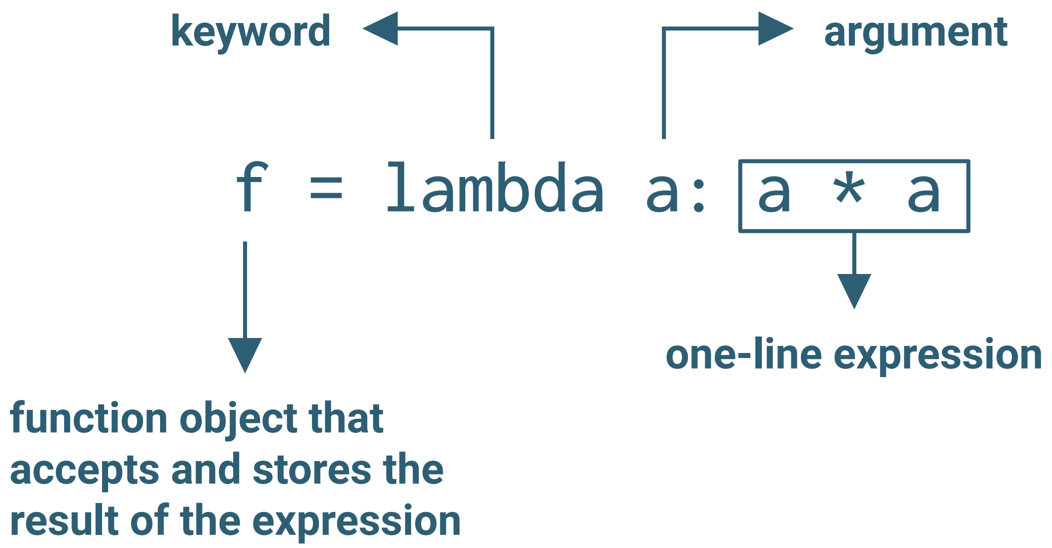


```
map_function = map(divide_by_seven,(24,654,3,961,21))
```

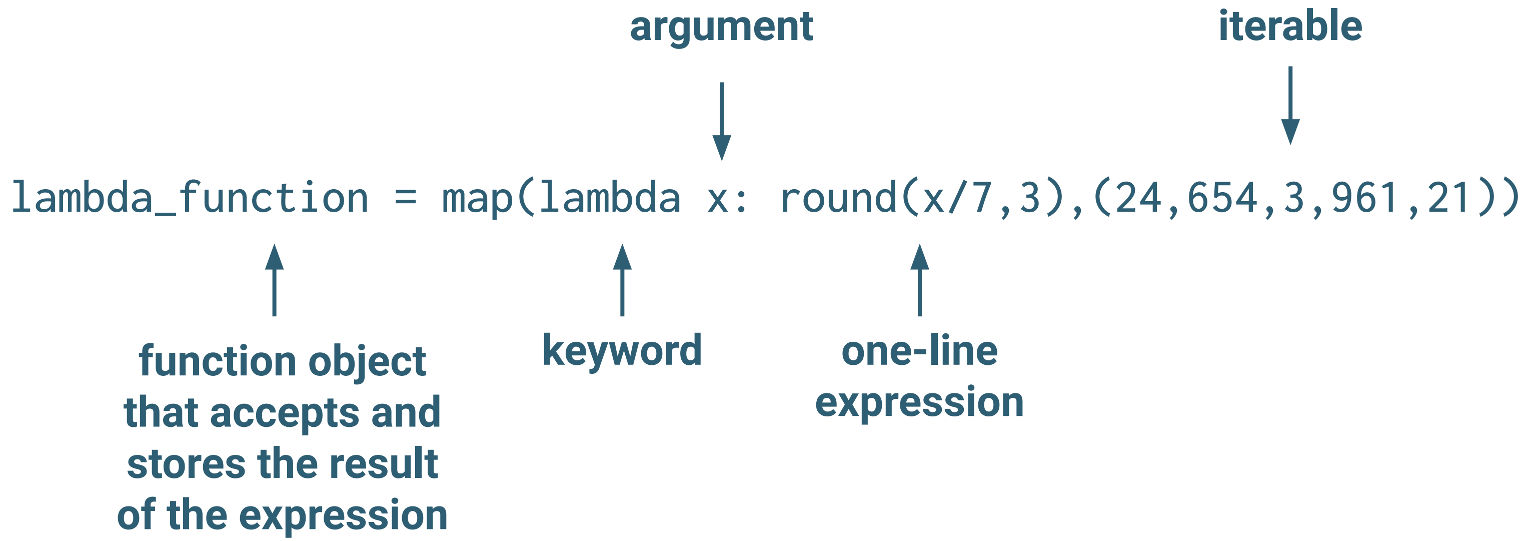
The `lambda` function

1. A `lambda` function is not stored as a defined function.
2. The `lambda` function is executed within the `map` function, and its output is returned just like a typical custom function.
3. `lambda` functions are defined using the reserved keyword `lambda`, followed by a parameter, a semi-colon, and then the code that defines the functionality.

The `lambda` function


keyword
argument
f = lambda a: a * a
one-line expression
function object that accepts and stores the result of the expression

The `lambda` function



```
lambda_function = map(lambda x: round(x/7,3),(24,654,3,961,21))
```

argument
iterable
function object that accepts and stores the result of the expression
keyword
one-line expression

The `filter` function

1. The `filter` function is most often used to filter out unwanted values from the input iterable.
2. The data returned from using the `filter` function will have certain values removed.
3. The `filter` function is most often used with the `lambda` function.

```
filter(function, iterable)
```

The `filter` function

```
# Use the filter and lambda functions to get only the numbers divided by 3.
numbers = [12, 7, 9, 18, 25, 36, 42, 55, 63]
divisible_by_3 = list(filter(lambda x: x % 3 == 0, numbers))
print(divisible_by_3
```

`filter(function, iterable)`

This is the "function": `lambda x: x % 3 == 0`

This is the "iterable": `[12,7,9,18,25,36,42,55,63]`

Activity:

Map, Lambda, and Filter Functions

In this activity, you’ll practice using the `map`, `lambda`, and `filter` functions.

Suggested Time:
10 Minutes

Time’s up!

Let’s review

Questions?

Instructor **Demonstration**

Documenting Your Code

Commenting Code

1. Commenting code is used to explain the purpose or functionality of the code underneath the comment.
2. Comments can made above the code or in-line with the code.
3. Comments can be on one line or multiple lines.

```
# Define a function that will add two numbers.
def add():
    first_number = 1 # This is a local scope of the function.
    second_number = 2 # This is a local scope of the function.
    total = first_number + second_number
    print("Your total is: ", total)

add()
```

```
# Use the filter and lambda functions to get
# only the numbers divided by 3.
numbers = [12, 7, 9, 18, 25, 36, 42, 55, 63]
divisible_by_3 = list(filter(lambda x: x % 3 ==
0, numbers))
print(divisible_by_3)
```

Common Symbols Used For Commenting Code:

Commenting Code Symbol	Programming Language
# <comment>	Python
// <comment>	JavaScript
<!-- <comment -->	Markdown
<!-- <comment -->	HTML
/* <comment */	CSS

Documentation

01

Documentation is meant to give users a **better understanding** of the functionality of your code.

02

Documentation is used to **explain** what functions and classes do.

03

When we document functions and classes, we are creating a **“docstring”**. To create a docstring we use the open and closed triple quotes (“”” “””).

Docstring Example

```
def my_function(parameter1, parameter2):
    """
    Brief description of the function.
   Optional:
     More detailed description of what the function does.
    Args:
        arg1 (type): Description of arg1.
        arg2 (type): Description of arg2.
    Returns:
        type: Description of the return value or print statement.
    Raises:
        ErrorType: Description of the exception raised, if any.
    Examples:
        Provide some usage examples of the function.
    Note:
        Any additional notes about the function.
    """
    # Function code here

if __name__ == "__main__":
    my_function(arg1=parameter1, arg2=parameter2)
```

Activity:

Pizza Order Documentation

In this activity, you'll practice adding a docstring that describes what a function does, and add one line documentation to explain what the code does.

Suggested Time:

10 Minutes

Time’s up!

Let’s review

Let’s **recap**

Review the Class Objective

In this lesson you learned how to:

1. Understand the importance of functions.
2. Create, write, and use Python functions.
3. Perform transformations on iterable objects using Python built-in functions.
4. Document and style code using the latest standards.

Next

In the next lesson, you'll learn to import modules and use their functions and methods, refactor code to make it cleaner, and learn how to take business and user needs into consideration when coding a solution to the problem at hand.

Questions?

The End