Functions

Unit 4

Introduction

In Python, a function is a block of organized, reusable code that is written to carry out a specified task.

And to carry out that specific task, the function might or might not need multiple inputs.

When the task is carried out, the function can or can not return one or more values.

This block of code will only executed when it is called.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability
- Functions allow you to break down complex problems into smaller, manageable pieces.(Modularity)
- With functions, updating or fixing code is easier.(Maintainability)
- Functions help in managing the scope of variables(Scope Management)
- Recursion
- Abstraction

There are three types of functions in Python:

- Built-in functions, such as help() to ask for help, min() to get the minimum value, print() to print an object to the terminal,… You can find an overview with more of these functions here.
- User-Defined Functions (UDFs), which are functions that users create to help them out; And
- Anonymous functions, which are also called lambda functions because they are not declared with the standard def keyword.

Functions vs Methods

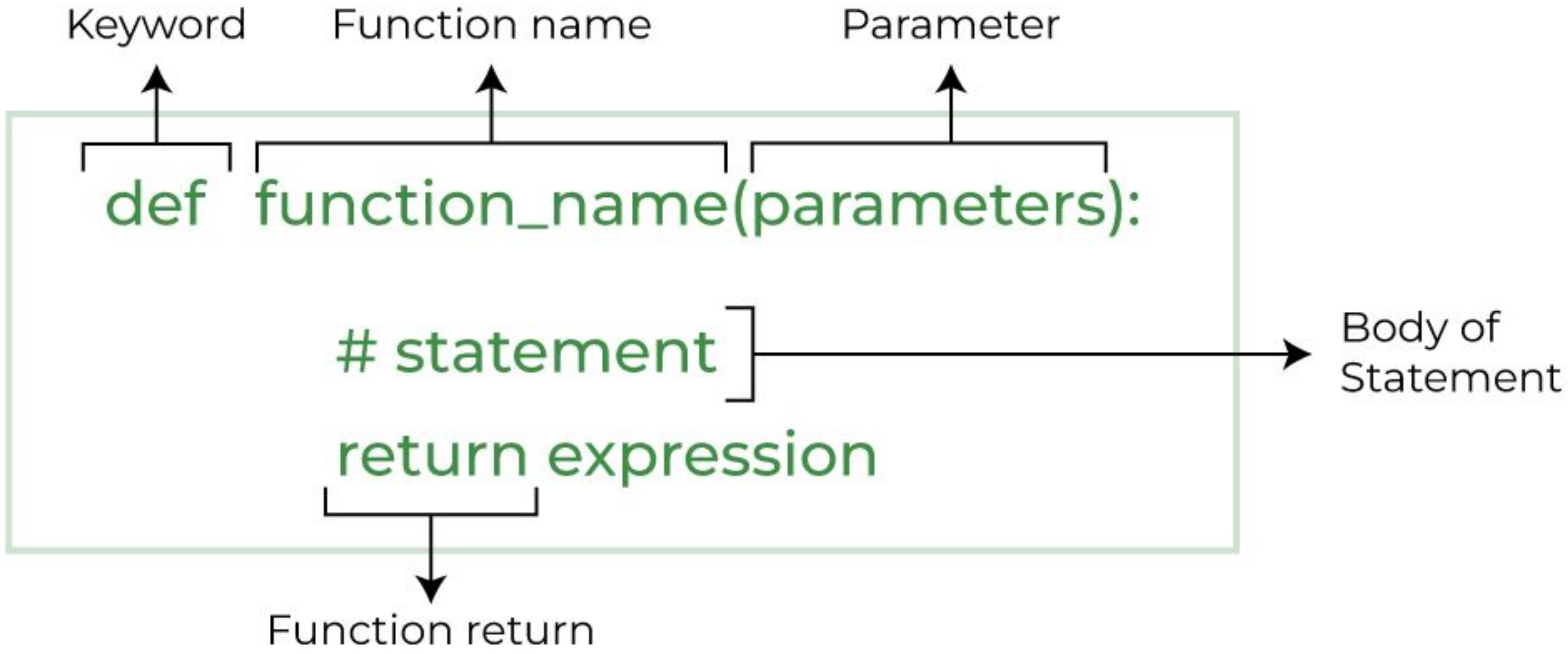
- A method refers to a function which is part of a class. You access it with an instance or object of the class.
- A function doesn’t have this restriction: it just refers to a standalone function. This means that all methods are functions, but not all functions are methods.

How to Define a Function: User-Defined Functions

The four steps to defining a function in Python are the following:

- Use the keyword def to declare the function and follow this up with the function name.
- Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.
- Add statements that the functions should execute.
- End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

The syntax to declare a function

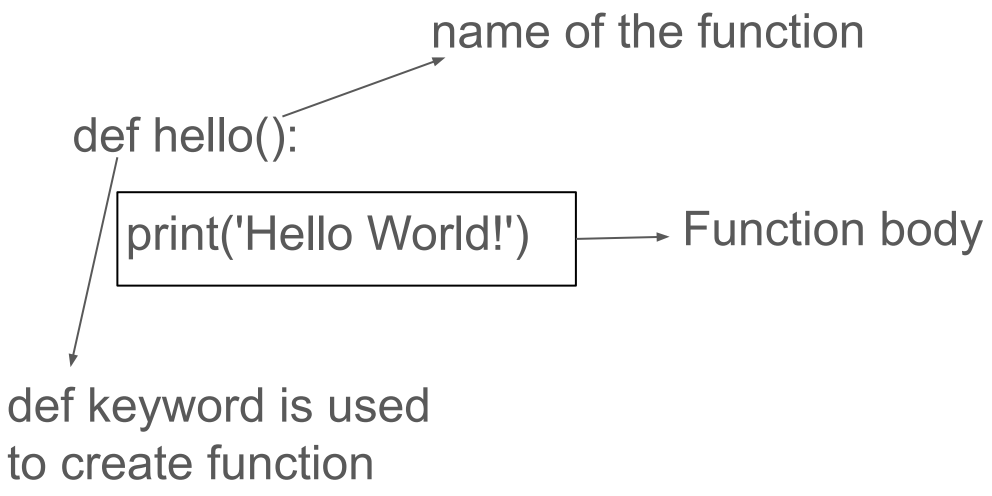


```
def function_name(parameters):

        # statement
        return expression
```

example

```
def hello():
    print('Hello World!')
```

Anatomy



```
def hello():
    print('Hello World!')
```

Here, we have created a simple function named hello() that prints Hello World!

Note: When writing a function, pay attention to indentation, which are the spaces at the start of a code line.

Calling a Function

In the above example, we have declared a function named hello().

```
def hello():
    print('Hello World!')
```

If we run the above code, we won't get an output.

It's because creating a function doesn't mean we are executing the code inside it. It means the code is there for us to use if we want to.

To use this function, we need to call the function.

Example: Function Call

```
def hello():
    print('Hello World!')


# call the function
hello()


print('Outside function')
```

Output

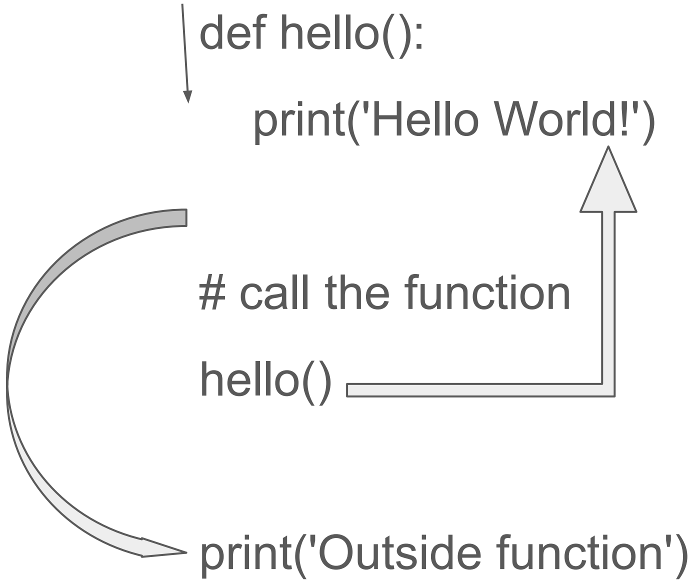
```
Hello World!

Outside function
```

Understand the flow

When the function hello() is called, the program's control transfers to the function definition.

All the code inside the function is executed.

The control of the program jumps to the next statement after the function call.



```
def hello():
    print('Hello World!')

# call the function
hello()

print('Outside function')
```

Passing Arguments

We can pass arguments to functions to provide them with input data necessary for their execution.

There are several ways to pass arguments to functions, each with its own use cases and advantages.

Positional Arguments

These are the most straightforward type of arguments. The values are passed to the function in the same order as the parameters are defined.

```
def hello(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")


hello("Ram", "Bahadur")
```

Output: Hello, Ram Bahadur!

Keyword Arguments

Keyword arguments are passed to the function by explicitly naming each parameter and assigning a value.

This method enhances readability and allows you to pass arguments in any order.

Example

```
def hello(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")


hello(first_name="Ram", last_name="Bahadur")

hello(last_name="Bahadur", first_name="Ram")
```

Output: Hello, Ram Bahadur!

Output: Hello, Ram Bahadur!

Default Arguments

You can assign default values to parameters. If an argument is not provided, the default value is used.

```
def hello(name="NCCS"):
    print(f"Hello, {name}!")


hello()
hello("Kakashi")
```

Output: Hello, NCCS!

Hello, Kakashi!

Packing and unpacking arguments (Arbitrary Keyword Arguments)

In Python, packing allows a function to accept an arbitrary number of arguments using the asterisk (*) syntax, while unpacking enables the distribution of a sequence's elements into multiple variables or function parameters.

There are two special syntaxes in Python that allow a function to receive a variable number of arguments:

- *args (or “variable-length positional arguments”): This syntax allows a function to receive any number of positional arguments. The arguments are passed to the function as a tuple.
- **kwargs (or “variable-length keyword arguments”): This syntax allows a function to receive any number of keyword arguments. The arguments are passed to the function as a dictionary.

Packing arguments

It refers to aggregating multiple arguments in a single collection so that the operations can be applied to the arguments combined. It is helpful when the same operation is to be called with a different number of parameters.

Unpacking arguments

It refers to deconstructing a collection into multiple arguments so that the operations can be applied to the arguments individually. It is helpful when the arguments are required separately to perform operations.

* operator to “pack” arguments into a single tuple

```
def pack_args(*args):
    print(type(args))
    for arg in args:
        print(arg)


pack_args(1, 2, 3, "Hello", [1, 2, 3])
```

Output

```
<class 'tuple'="">

1

2

3

Hello

[1, 2, 3]
```

In this example, the pack_args function takes a variable number of arguments using the * operator.

This operator allows the function to accept any number of arguments, which are then packed into a single tuple named args.

The type of args is tuple, and the elements can be accessed individually, as demonstrated by the for-loop.

** operator to “pack” keyword arguments into a single dictionary

```
def pack_kwargs(**kwargs):
    print(type(kwargs))
    for key, value in kwargs.items():
        print(f"{key}: {value}")
pack_kwargs(name="Ram", age=30, country="Nepal")
```

Output

```
<class 'dict'="">

name: Ram

age: 30

country: Nepal
```

What are Unpacking Arguments?

Unpacking arguments in Python refers to the process of taking a packed tuple or dictionary and “unpacking” its elements into separate variables. The * and ** operators can be used to unpack the arguments, respectively.

unpack a tuple of positional arguments using the * operator:

```
def add(a, b, c):

    print(a + b + c)


#Call the function with a packed tuple of arguments

args = (1, 2, 3)

add(*args)
```

Output

```
6
```

unpack a dictionary of keyword arguments using the ** operator

```
def print_person(name, age, country):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Country: {country}")

#Call the function with a packed dictionary of keyword arguments
person = {"name": "Ram", "age": 30, "country": "Nepal"}
print_person(**person)
```

Output

```
Name: Ram

Age: 30

Country: Nepal
```

Working of Packing and Unpacking Arguments Together

```
def add_subtract(a, b, c, *args, **kwargs):
    result = a + b - c
    for arg in args:
        result += arg
    for key, value in kwargs.items():
        result += value
    return result
numbers = (1, 2, 3)
more_numbers = (4, 5, 6)
additional_numbers = {"x": 7, "y": 8, "z": 9}
print(add_subtract(*numbers,*more_numbers,**additional_numbers))
```

Output

```
39
```

What will be the output

```
def showData(a, b, c):

print(a, b, c)



myData = (1, 2, 3)

showData(myData)
```

```
def showData(a, b, c):
    print(a, b, c)


myData = (1, 2)
showData(*myData)
```

```
TypeError: showData() missing 1 required positional argument: 'c'
```

```
def showData(a, b, *args):
    print(a, b, args)


myData = (1, 2, 3, 4, 5)
showData(*myData)
```

1, 2, (3, 4, 5)

Passing List as argument

```
def fruits_function(food):
  for x in food:
    print(x)


fruits = ["apple", "banana", "cherry"]


my_function(fruits)
```

output

```
apple

banana

cherry
```

Return Values

In Python, functions can return values using the return statement. Python also allows functions to return multiple values in various ways.

Returning a Single Value

A function in Python can return a single value, which can be of any data type (e.g., int, float, string, list, object).

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


result = add(3, 5)
print(result)
```

Output

```
8
```

Returning Multiple Values

Python supports returning multiple values from a function using tuples, lists, or dictionaries. Here are a few methods:

Using Tuples

```
def get_name_and_age():
    name = "Ram"
    age = 30
    return name, age  # Returning a tuple

name, age = get_name_and_age()
print(name)
print(age)
```

Output

```
Alice

30
```

Using Dictionaries

```
def get_person_info():
    return {"name": "Ram", "age": 25, "city": "kathmandu"}

person_info = get_person_info()
print(person_info)
print(person_info['name'])
print(person_info['age'])
print(person_info['city'])
```

Output

```
{'name': 'Ram', 'age': 25, 'city': 'Kathmandu'}

Ram

25

Kathmandu
```

what happens when we don't put return at the end of function?

In Python, if a function does not have a return statement, or if the return statement is omitted, the function will return None by default when it reaches the end.

Function Without return Statement

When a function does not include a return statement, it implicitly returns None.

```
def test(name):
    print(f"Hello, {name}!")


result = test("Alice")
print(result)
```

Output: None

Function with an Empty return Statement

A function can have an empty return statement, which also results in returning None.

```
def test(name):
    print(f"Hello, {name}!")
    return


result = test("Alice")
print(result)
Output: None
```

Recursive Function

Recursion

Recursion is the process of defining something in terms of itself.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

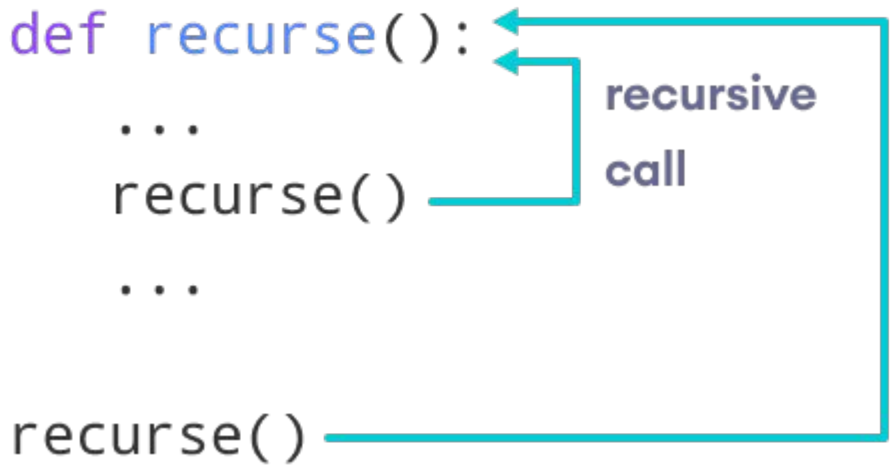
- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Recursive Function

We all know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

In simple words, recursive function is a function which calls itself is directly or indirectly is known as recursive function.

syntax



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Example of a recursive function

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```

In the above example, factorial() is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)            # 1st call with 3

3 * factorial(2)        # 2nd call with 2

3 * 2 * factorial(1)    # 3rd call with 1

3 * 2 * 1               # return from 3rd call as number=1

3 * 2                   # return from 2nd call

6                       # return from 1st call
```

Remember

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.

```
def recursor():
    recursor()
recursor()
```

Output

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Lambda Function

Introduction to Anonymous Functions

Anonymous functions are functions that are defined without a name.

In Python, these are created using the lambda keyword.

Unlike regular functions defined with def, anonymous functions are typically used for short-term operations and do not need a name.

When to Use a Anonymous Function in Python

Although you can use both regular functions and lambda functions to achieve the same results, here are some of the reasons why you might pick a lambda function:

- First of all, you can use a lambda function when you need a function that'll be used just once. This is especially useful when working with functions like map, reduce, filter.
- You can also use a lambda function when you need a function that should be invoked immediately.
- Finally, lambdas are useful when you want to use a function inside a function. Or when you want to create a function that returns a function.

Introduction

Lambda Functions in Python are anonymous functions, implying they don't have a name.

The def keyword is needed to create a typical function in Python, as we already know.

We can also use the lambda keyword in Python to define an unnamed function.

Basic syntax of a lambda function

lambda arguments: expression

- lambda is the keyword.
- arguments are the inputs to the function.
- expression is what the function returns.

Examples

Adding Two Numbers

```
add = lambda x, y: print(x + y)

add(2, 3)
```

```
Output: 5
```

```
add = lambda x, y: x + y

print(add(2, 3))
```

```
Output: 5
```

```
(lambda x, y: print(x + y)) (2, 3)
```

```
Output: 5
```

Squaring a Number

```
square = lambda x : x**2

print(square(5))
```

```
Output: 25
```

Comparing with Regular Functions

Regular Function

```
def add(x, y):
    return x + y
print(add(2, 3))
```

Output: 5

Lambda Function

```
add = lambda x, y: x + y
print(add(2, 3))
```

```
Output: 5
```

Use Cases of Lambda Functions

Using with Higher-Order Functions:

Higher-order functions are functions that take other functions as arguments or return them as results. Lambda functions are particularly useful when working with higher-order functions because they provide a concise way to define the functions needed.

map()

The map() function is a built-in Python function that applies a given function to all items in an input iterable (such as a list, tuple, etc.) and returns a map object (an iterator).

The syntax for map() is:

map(function, iterable, ...)

function: The function to be applied to each item in the iterable.

iterable: One or more iterables whose items will be processed by the function.

How map() Works

The map() function returns an iterator that yields results of applying the function to the items of the iterable(s).

If multiple iterables are passed, the function must accept as many arguments as there are iterables, and the items are processed in parallel.

Single Iterable

Squaring Numbers

```
numbers = [1, 2, 3, 4]

squared = map(lambda x: x ** 2, numbers)

print(list(squared))
```

```
Output: [1, 4, 9, 16]
```

Multiple Iterables

```
list1 = [1, 2, 3]

list2 = [4, 5, 6]

added = map(lambda x, y: x + y, list1, list2)

print(list(added))
```

```
Output: [5, 7, 9]
```

filter()

The filter() function is a built-in Python function that constructs an iterator from elements of an iterable for which a function returns True. Essentially, it filters out items that do not meet a certain condition.

The syntax for filter() is:

filter(function, iterable)

function: A function that tests each element of the iterable. It should return True or False.

iterable: The iterable whose elements are to be filtered.

How filter() Works

The filter() function applies the given function to each item of the iterable and returns an iterator with elements that return True when passed to the function.

Example

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even = filter(lambda x: x % 2 == 0, numbers)

print(list(even))

Output: [2, 4, 6, 8, 10]
```

sorted()

The sorted() function is a built-in Python function that returns a new sorted list from the elements of any iterable. Unlike the sort() method, which modifies the list in place, sorted() creates a new sorted list without changing the original iterable.

The syntax for sorted() is:

- sorted(iterable, key=None, reverse=False)

iterable: The sequence (such as list, tuple, dictionary, etc.) that you want to sort.

key (optional): A function that serves as a key for the sort comparison. The default value is None.

reverse (optional): A boolean value. If True, the list elements are sorted as if each comparison were reversed. The default value is False.

How sorted() Works

The sorted() function sorts the elements of the iterable based on the provided key function and reverse flag.

Example: Sorting a List of Numbers

```
numbers = [5, 2, 9, 1, 5, 6]

sorted_numbers = sorted(numbers, key=lambda x: x)

print(sorted_numbers)
```

Output: [1, 2, 5, 5, 6, 9]

Sorting in Descending Order

```
numbers = [5, 2, 9, 1, 5, 6]

sorted_numbers_desc = sorted(numbers, key=lambda x: -x)

print(sorted_numbers_desc)
```

```
Output: [9, 6, 5, 5, 2, 1]
```

Sorting a List of Tuples

Sorting by the Second Element

```
pairs = [(1, 2), (3, 1), (5, 0)]

sorted_pairs = sorted(pairs, key=lambda x: x[1])

print(sorted_pairs)
```

```
Output: [(5, 0), (3, 1), (1, 2)]
```

Sorting by the First Element

```
pairs = [(1, 2), (3, 1), (5, 0)]

sorted_pairs = sorted(pairs, key=lambda x: x[0])

print(sorted_pairs)
```

```
Output:[(1, 2), (3, 1), (5, 0)]
```

OR

```
pairs = [(1, 2), (3, 1), (5, 0)]

sorted_pairs = sorted(pairs, key=lambda x: x)

print(sorted_pairs)
```

```
Output: [(1, 2), (3, 1), (5, 0)]
```

Sorting a List of Dictionaries

```
students = [
    {"name": "Ram", "grade": 5},
    {"name": "Hari", "grade": 9},
    {"name": "Shyam", "grade": 1}
]
sorted_students = sorted(students, key=lambda x: x['grade'])
print(sorted_students)
```

```
Output: [{'name': 'Shyam', 'grade': 1}, {'name': 'Ram', 'grade': 5}, {'name': 'Hari',
'grade': 9}]
```

Exercise

Exercise 1: Write a lambda function that multiplies two numbers.

Exercise 2: Use a lambda function inside map to convert a list of temperatures from Celsius to Fahrenheit.

Exercise 3: Use a lambda function inside filter to extract words longer than 4 characters from a list of words.