Python Higher Order Functions

Introduction

A **higher order function** is a function that works with other functions.

A function can be treated like a value in Python.

That means a function can be:

- stored in a variable
- passed as an argument
- returned from another function
- used inside another function

This idea is extremely important because it allows us to build flexible and reusable programs.

For example:

```
def double(x):
    return x * 2

def apply_operation(value, operation):
    return operation(value)

result = apply_operation(5, double)

print(result)
```

Output:

```
10
```

Here:

```
apply_operation()
```

is a **higher order function** because it receives another function as an argument.

---

1. What Does "Higher Order" Mean?

A higher order function is a function that does at least one of these:

1. **Accepts another function as an argument**
2. **Returns another function as its result**

For example:

```
def apply_function(value, function):
    return function(value)
```

`apply_function()` accepts:

```
function
```

as an argument.

Therefore, it is a higher order function.

---

2. Functions Are Values in Python

In Python, functions are objects.

That means you can assign a function to a variable.

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

message = greet

message()
```

Output:

```
Hello
```

Notice:

```
message = greet
```

There are no parentheses after `greet`.

We are storing the function itself.

---

3. Function vs Function Call

This distinction is very important.

Function itself

```
greet
```

This refers to the function.

Function call

```
greet()
```

This executes the function.

Consider:

```
def greet():
    return "Hello"

x = greet
y = greet()
```

Here:

```
x
```

contains the function.

While:

```
y
```

contains:

```
"Hello"
```

because the function was executed.

---

4. Passing a Function as an Argument

A function can receive another function.

```
def square(x):
    return x * x

def calculate(value, operation):
    return operation(value)

result = calculate(5, square)

print(result)
```

Output:

```
25
```

The flow is:

```
square
   ↓
```

```
passed into calculate()
   ↓
operation(value)
   ↓
square(5)
   ↓
25
```

---

5. A Simple Higher Order Function

```
def double(x):
    return x * 2

def apply_operation(number, operation):
    return operation(number)

print(apply_operation(10, double))
```

Output:

```
20
```

The important part is:

```
operation(number)
```

The function received through `operation` is called.

---

6. Passing Different Functions

The real power becomes clear when we pass different functions.

```
def double(x):
    return x * 2

def square(x):
    return x * x

def apply_operation(number, operation):
    return operation(number)

print(apply_operation(5, double))
print(apply_operation(5, square))
```

Output:

```
10
25
```

We did not need separate functions like:

```
apply_double()
apply_square()
```

Instead, one function can work with different operations.

---

7. Using Lambda With Higher Order Functions

Higher order functions work especially well with lambda functions.

```
def apply_operation(value, operation):
    return operation(value)

result = apply_operation(
    5,
    lambda x: x * 10
)

print(result)
```

Output:

```
50
```

Here:

```
lambda x: x * 10
```

is passed as an argument.

---

8. Higher Order Functions With `map()`

Python's `map()` is a built-in higher order function.

It receives:

1. a function
2. an iterable

Example:

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

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

print(list(result))
```

Output:

```
[2, 4, 6, 8]
```

`map()` receives the lambda function and applies it to every item.

---

9. Understanding `map()` Step by Step

Given:

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

and:

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

`map()` effectively performs:

```
1 → 1 × 2 → 2
2 → 2 × 2 → 4
3 → 3 × 2 → 6
```

Result:

```
[2, 4, 6]
```

This is an example of passing behavior into another function.

---

10. Higher Order Functions With `filter()`

`filter()` is another higher order function.

It receives a function that decides whether each item should remain.

```
numbers = [1, 2, 3, 4, 5, 6]

result = filter(
    lambda x: x % 2 == 0,
```

```
    numbers
)

print(list(result))
```

Output:

```
[2, 4, 6]
```

The lambda returns:

```
True
```

for even numbers.

---

11. Higher Order Functions With `sorted()`

`sorted()` can also receive a function through the `key` parameter.

```
students = [
    ("Hafsa", 85),
    ("Asim", 92),
    ("Ali", 78)
]

result = sorted(
    students,
    key=lambda student: student[1]
)

print(result)
```

Output:

```
[('Ali', 78), ('Hafsa', 85), ('Asim', 92)]
```

The lambda tells `sorted()` what value to use when comparing items.

---

12. Functions Can Be Returned

Higher order functions can also **return another function**.

Example:

```
def create_greeting():
    def greet():
```

```
        return "Hello"

    return greet

message = create_greeting()

print(message())
```

Output:

```
Hello
```

Here:

```
create_greeting()
```

returns the `greet` function.

---

13. Returning a Lambda

The same idea can be written using lambda.

```
def create_multiplier(number):
    return lambda x: x * number

double = create_multiplier(2)
triple = create_multiplier(3)

print(double(5))
print(triple(5))
```

Output:

```
10
15
```

The function creates another function based on the supplied value.

---

14. Why Return a Function?

Suppose we need several multipliers.

Without higher order functions:

```
def double(x):
    return x * 2
```

```
def triple(x):
    return x * 3

def quadruple(x):
    return x * 4
```

This works, but we are repeating the same structure.

With a higher order function:

```
def create_multiplier(multiplier):
    return lambda x: x * multiplier
```

Now:

```
double = create_multiplier(2)
triple = create_multiplier(3)
quadruple = create_multiplier(4)
```

One function can create many specialized functions.

---

15. Functions as Data

Think about functions like other Python values.

You can do:

```
name = "Hafsa"
number = 10
```

You can also do:

```
def greet():
    return "Hello"

function = greet
```

Now `function` refers to the same function as `greet`.

This makes functions **first-class objects** in Python.

---

16. First-Class Functions

When a programming language treats functions as values that can be:

- assigned to variables
- passed as arguments
- returned from functions

- stored in data structures

we say that functions are **first-class objects**.

Python supports this.

Example:

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

operation = add

print(operation(5, 3))
```

Output:

```
8
```

---

17. Storing Functions in a List

Because functions are objects, they can be stored in collections.

```
def add_one(x):
    return x + 1

def double(x):
    return x * 2

def square(x):
    return x * x

operations = [add_one, double, square]

print(operations[0](5))
print(operations[1](5))
print(operations[2](5))
```

Output:

```
6
10
25
```

This can be useful when you want to select an operation dynamically.

---

18. Selecting a Function Dynamically

For example:

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

operations = {
    "add": add,
    "subtract": subtract,
    "multiply": multiply
}

operation = operations["multiply"]

print(operation(5, 4))
```

Output:

```
20
```

The dictionary stores functions as values.

---

19. Practical Example: Calculator

We can build a flexible calculator using higher order functions.

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def calculate(a, b, operation):
    return operation(a, b)
```

```
print(calculate(10, 5, add))
print(calculate(10, 5, subtract))
print(calculate(10, 5, multiply))
```

Output:

```
15
5
50
```

The `calculate()` function does not need to know which operation will be used.

It simply receives the operation.

---

20. Practical Example: haas.dev Resource Processing

Imagine haas.dev has a list of resource titles:

```
resources = [
    "Python Functions",
    "Python Lists",
    "Python Dictionaries"
]
```

We can create a reusable processing function:

```
def process_resources(resources, operation):
    return [
        operation(resource)
        for resource in resources
    ]
```

Now we can pass different operations.

```
uppercase = process_resources(
    resources,
    lambda resource: resource.upper()
)

print(uppercase)
```

Output:

```
['PYTHON FUNCTIONS', 'PYTHON LISTS', 'PYTHON DICTIONARIES']
```

We could pass another function:

```
def add_prefix(resource):
    return "haas.dev: " + resource

result = process_resources(
    resources,
    add_prefix
)

print(result)
```

The same `process_resources()` function works with different behavior.

---

21. Practical Example: Resource Filtering

We can also create a reusable filtering function.

```
def filter_resources(resources, condition):
    return [
        resource
        for resource in resources
        if condition(resource)
    ]
```

Use it like this:

```
resources = [
    {"title": "Python", "views": 500},
    {"title": "JavaScript", "views": 800},
    {"title": "DSA", "views": 300}
]

popular = filter_resources(
    resources,
    lambda resource: resource["views"] >= 500
)

print(popular)
```

Output:

```
[
    {'title': 'Python', 'views': 500},
    {'title': 'JavaScript', 'views': 800}
]
```

The filtering function does not know the specific rule.

The caller provides the rule.

---

22. The Key Idea: Separate Logic From Behavior

Consider:

```
def process(value, operation):
    return operation(value)
```

The function knows **how to process** the value.

But it does not know **what operation** should happen.

That behavior is supplied from outside.

This creates flexible code.

```
Data
 ↓
Higher Order Function
 ↓
Provided Operation
 ↓
Result
```

---

23. Higher Order Functions and Abstraction

Higher order functions allow us to hide repetitive implementation details.

Instead of writing:

```
def double_all(numbers):
    return [x * 2 for x in numbers]

def square_all(numbers):
    return [x * x for x in numbers]

def add_ten_all(numbers):
    return [x + 10 for x in numbers]
```

we can create:

```
def transform_all(numbers, operation):
    return [operation(x) for x in numbers]
```

Then:

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

double = transform_all(numbers, lambda x: x * 2)
square = transform_all(numbers, lambda x: x * x)
add_ten = transform_all(numbers, lambda x: x + 10)
```

One function handles the common structure.

---

24. `map()`, `filter()`, and `sorted()` Are Examples

You have already seen several Python functions that use this concept.

`map()`

```
map(function, iterable)
```

`filter()`

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

`sorted()`

```
sorted(data, key=function)
```

They accept functions as arguments.

That makes them examples of higher order programming.

---

25. A Function That Returns a Function

Let's build a discount calculator.

```
def create_discount(rate):

    def apply_discount(price):
        return price - (price * rate)

    return apply_discount
```

Create different discount functions:

```
student_discount = create_discount(0.10)
premium_discount = create_discount(0.20)

print(student_discount(1000))
print(premium_discount(1000))
```

Output:

```
900.0
800.0
```

The outer function creates customized functions.

---

26. Closures

The previous example introduces another important concept: a **closure**.

```
def create_discount(rate):

    def apply_discount(price):
        return price - (price * rate)

    return apply_discount
```

After `create_discount()` finishes, the returned function still remembers:

```
rate
```

This behavior is called a closure.

Closures are an advanced topic, but understanding higher order functions makes them easier to learn later.

---

27. Higher Order Functions vs Normal Functions

Normal function

```
def square(x):
    return x * x
```

It receives data.

```
number → function → result
```

Higher order function

```
def apply(value, operation):
    return operation(value)
```

It receives:

```
data + behavior
```

The behavior is represented by another function.

---

28. Common Mistake: Calling Instead of Passing

Incorrect:

```
calculate(5, double())
```

This executes `double()` immediately.

Usually you want:

```
calculate(5, double)
```

Here you pass the function itself.

Remember:

```
double
   → function

double()
   → function result
```

---

29. Common Mistake: Returning the Result Instead of the Function

Suppose we want to return a function.

Incorrect:

```
def create():
    return greet()
```

This returns the result of `greet()`.

To return the function itself:

```
def create():
    return greet
```

The difference is:

```
greet   → function
greet() → result
```

---

30. Common Mistake: Overusing Higher Order Functions

Higher order functions are powerful, but not every problem needs them.

This:

```
result = apply_operation(
    10,
    lambda x: x * 2
)
```

is reasonable.

But if the operation is complex:

```
result = apply_operation(
    data,
    lambda x: complicated_expression_here
)
```

a named function may be much clearer.

Use abstraction when it improves the design, not simply because Python allows it.

---

31. Higher Order Functions and Readability

Compare:

```
result = process(
    data,
    lambda x: x["price"] * x["quantity"] if x["active"] else 0
)
```

with:

```
def calculate_item_total(item):
    if item["active"]:
        return item["price"] * item["quantity"]

    return 0

result = process(data, calculate_item_total)
```

The second version is longer, but the behavior has a meaningful name.

Choose the version that makes the program easier to understand.

---

32. When Should You Use Higher Order Functions?

They are useful when:

- the same process needs different behaviors
- you want reusable data processing
- you want to avoid repeated code
- you need configurable operations
- you are working with `map()`, `filter()`, or `sorted()`
- you want to pass behavior into another function
- you are building callbacks or event-driven logic

---

33. When Should You Avoid Them?

Avoid unnecessary higher order abstractions when:

- the problem is already simple
- the extra function layer makes the code harder to read
- a normal loop is clearer
- the function is only being passed around without providing real flexibility
- beginners reading the code would struggle to understand the abstraction

Good code is not the code with the most advanced features.

Good code solves the problem clearly.

---

34. Problem Solving Framework

When you see repeated code, ask:

Step 1: What part is always the same?

For example:

```
Loop through every resource.
```

Step 2: What part changes?

For example:

```
Convert title to uppercase.
Add a prefix.
Calculate length.
```

Step 3: Can the changing behavior become a function?

For example:

```
lambda resource: resource.upper()
```

Step 4: Can the common structure receive that function?

```
def process_resources(resources, operation):
    return [operation(resource) for resource in resources]
```

Step 5: Does this make the code clearer?

If yes, use the abstraction.

If no, keep the simpler solution.

---

35. Mini Project: Flexible Calculator

Build a calculator that accepts operations dynamically.

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Cannot divide by zero"

    return a / b


def calculate(a, b, operation):
    return operation(a, b)


print(calculate(20, 5, add))
print(calculate(20, 5, subtract))
print(calculate(20, 5, multiply))
print(calculate(20, 5, divide))
```

Output:

```
25
15
```

```
100
4.0
```

What makes `calculate()` higher order?

It receives:

```
operation
```

which is another function.

---

36. Mini Project: Resource Processor

Create a reusable processor:

```
def process_resources(resources, operation):
    results = []

    for resource in resources:
        results.append(operation(resource))

    return results
```

Now use different operations:

```
resources = [
    "Python",
    "JavaScript",
    "React"
]

uppercase = process_resources(
    resources,
    lambda resource: resource.upper()
)

lengths = process_resources(
    resources,
    lambda resource: len(resource)
)

print(uppercase)
print(lengths)
```

Output:

```
['PYTHON', 'JAVASCRIPT', 'REACT']
[6, 10, 5]
```

One processing function supports different behaviors.

---

37. 5 Minute Challenge

Create:

```
numbers = [2, 4, 6, 8, 10]
```

Then create:

```
def process_numbers(numbers, operation):
    ...
```

Use it to:

1. Double every number.
2. Square every number.
3. Add 5 to every number.
4. Convert every number to a string.

Try to solve all four using the same `process_numbers()` function.

---

38. Exercises

Exercise 1

Create:

```
def apply_operation(value, operation):
    ...
```

Use it with:

```
lambda x: x * 5
```

Exercise 2

Create two functions:

```
add_one()
square()
```

Then create a higher order function that accepts either function.

Exercise 3

Use `map()` and a lambda to double:

```
[5, 10, 15, 20]
```

Exercise 4

Use `filter()` and a lambda to keep numbers greater than 50:

```
[20, 60, 40, 90, 30, 80]
```

Exercise 5

Create a function:

```
create_multiplier(number)
```

It should return another function that multiplies its argument by `number`.

Expected usage:

```
double = create_multiplier(2)

print(double(10))
```

Expected output:

```
20
```

Exercise 6

Create a function:

```
process_students(students, operation)
```

It should apply the supplied function to every student.

---

39. Mini Quiz

Question 1

What is a higher order function?

A. A function with many lines

B. A function that works with other functions

C. A function that only returns numbers

D. A function inside a class

Answer: B

Question 2

Which is an example of passing a function?

```
  calculate(10, double)
```

A. Yes

B. No

Answer: A

Question 3

What does this represent?

```
  double
```

A. The result of calling `double`

B. The function itself

C. A string

D. A number

Answer: B

Question 4

What does this represent?

```
  double()
```

A. The function itself

B. A function definition

C. The result of executing the function

D. A variable declaration

Answer: C

Question 5

Which Python functions commonly accept another function?

A. `map()`

B. `filter()`

C. `sorted()` with `key`

D. All of the above

Answer: D

Question 6

Can a function return another function?

A. Yes

B. No

Answer: A

---

40. Interview Questions

Beginner

1. What is a higher order function?
2. What does it mean that functions are first-class objects in Python?
3. How can you pass a function as an argument?
4. What is the difference between `function` and `function()`?
5. Can a function return another function?
6. Give an example of a built-in higher order function.

Intermediate

7. How does `map()` use a function?
8. How does `filter()` use a function?
9. How does `sorted()` use the `key` parameter?
10. What is a callback function?
11. What is a closure?
12. Why are higher order functions useful?
13. When would you use a named function instead of a lambda?
14. How can higher order functions reduce code duplication?

---

41. Cheat Sheet

Store a Function

```
def greet():
    return "Hello"
```

```
message = greet
```

Call Stored Function

```
print(message())
```

Pass Function as Argument

```
def apply(value, operation):
    return operation(value)
```

Use It

```
apply(5, lambda x: x * 2)
```

Return a Function

```
def create_multiplier(n):
    return lambda x: x * n
```

`map()`

```
map(lambda x: x * 2, numbers)
```

`filter()`

```
filter(lambda x: x > 10, numbers)
```

`sorted()`

```
sorted(data, key=lambda x: x["score"])
```

Core Concept

```
Function
   ↓
can be treated like a value
   ↓
can be passed around
   ↓
can be used by another function
```

---

42. Key Takeaways

- Python functions are **first-class objects**.

- A function can be stored in a variable.
- A function can be passed as an argument.
- A function can return another function.
- A function that accepts or returns another function is a **higher order function**.
- `map()`, `filter()`, and `sorted()` are common examples of higher order programming.
- Lambda functions are often used with higher order functions.
- Higher order functions help separate **common processing logic** from **changing behavior**.
- They can reduce repetition and make programs more flexible.
- They should be used when they improve the structure and readability of the program.
- Understanding higher order functions prepares you for concepts such as **callbacks, closures, decorators, and functional programming**.

Visit haas.dev for more resources and guides.

Website: https://dev-roast-app.vercel.app

Resources: https://dev-roast-app.vercel.app/resources