**Higher-Order Functions in Python**

**Introduction to Higher-Order Functions**

**Definition**: A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result. Higher-order functions are a key concept in functional programming, allowing functions to be used as first-class citizens.

**Benefits**:

* Promote code reuse and abstraction.
* Facilitate functional programming paradigms.
* Enable more expressive and concise code.

**Basic Examples of Higher-Order Functions**

**Functions as Arguments**

A higher-order function can take another function as an argument.

**Example**: map

def square(x):

return x \* x

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

squared\_numbers = map(square, numbers)

print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

In this example:

* The map function takes square (a function) and numbers (an iterable) as arguments and applies square to each element in numbers.

**Functions as Return Values**

A higher-order function can return another function.

**Example**: make\_adder

def make\_adder(n):

def adder(x):

return x + n

return adder

add\_five = make\_adder(5)

print(add\_five(10)) # Output: 15

In this example:

* make\_adder returns a new function adder that adds n to its argument x.
* add\_five is a function that adds 5 to its input.

**Common Higher-Order Functions in Python**

**map**

map applies a given function to each item of an iterable and returns an iterator.

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

squared\_numbers = map(lambda x: x \* x, numbers)

print(list(squared\_numbers)) # Output: [1, 4, 9, 16, 25]

**filter**

filter constructs an iterator from elements of an iterable for which a function returns true.

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

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

print(list(even\_numbers)) # Output: [2, 4]

**reduce**

reduce applies a function of two arguments cumulatively to the items of an iterable, reducing the iterable to a single value.

from functools import reduce

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

sum\_of\_numbers = reduce(lambda x, y: x + y, numbers)

print(sum\_of\_numbers) # Output: 15

**Custom Higher-Order Functions**

You can create custom higher-order functions to encapsulate common patterns or logic.

**apply\_twice**

A function that applies another function twice.

def apply\_twice(func, x):

return func(func(x))

def add\_one(x):

return x + 1

print(apply\_twice(add\_one, 5)) # Output: 7

**Composing Functions**

Function composition is the process of combining two or more functions to produce a new function.

**compose**

A function that composes two functions.

def compose(f, g):

return lambda x: f(g(x))

def double(x):

return x \* 2

def increment(x):

return x + 1

increment\_and\_double = compose(double, increment)

print(increment\_and\_double(5)) # Output: 12

In this example:

* compose creates a new function that applies g to its input and then applies f to the result.
* increment\_and\_double increments the input by 1 and then doubles the result.

**Partial Application and Currying**

Partial application and currying are techniques for fixing a number of arguments to a function, producing another function of fewer arguments.

**partial from functools**

partial allows you to fix a certain number of arguments of a function and generate a new function.

from functools import partial

def multiply(x, y):

return x \* y

double = partial(multiply, 2)

print(double(5)) # Output: 10

**Practical Examples**

**Sorting with Custom Key Functions**

Higher-order functions are often used in sorting with custom key functions.

students = [

{'name': 'Alice', 'grade': 'A'},

{'name': 'Bob', 'grade': 'B'},

{'name': 'Charlie', 'grade': 'C'}

]

sorted\_students = sorted(students, key=lambda student: student['grade'])

print(sorted\_students)

# Output: [{'name': 'Alice', 'grade': 'A'}, {'name': 'Bob', 'grade': 'B'}, {'name': 'Charlie', 'grade': 'C'}]

**Event Handling**

Higher-order functions are useful in event-driven programming, such as setting up event handlers.

def on\_event(event\_handler):

def wrapper(event):

print(f"Handling event: {event}")

return event\_handler(event)

return wrapper

@on\_event

def handle\_click(event):

print(f"Clicked at {event['x']}, {event['y']}")

click\_event = {'x': 10, 'y': 20}

handle\_click(click\_event)

# Output:

# Handling event: {'x': 10, 'y': 20}

# Clicked at 10, 20

**Summary**

* **Higher-Order Functions**: Functions that take other functions as arguments or return functions as results.
* **Common Examples**: map, filter, reduce, and custom higher-order functions.
* **Function Composition**: Combining functions to create new functions.
* **Partial Application and Currying**: Techniques for creating functions with fewer arguments.
* **Practical Applications**: Sorting with custom keys, event handling, and more.

Higher-order functions are a fundamental aspect of functional programming in Python, enabling powerful abstractions and concise, expressive code.