Lecture 04 - Functions

Overview

Functions are a key concept in programming that allow to reuse code, make programs more modular, and simplify complex tasks.

In Python, functions are defined using the `def` keyword.

This notebook covers:

- **Definition** and **calls**
- **Parameters** and **arguments**
- **Return values**
- **Scope** of variables
- **Built-in** vs. **user-defined** functions
- **Functional** and **anonymous** programming

1. Basics of Functions

1.1 Definition and calls

In Python, a function is **defined** using the `def` keyword followed by the function name and parentheses `()`.

Once implemented, the function is called using the name and parantheses.

As long as the function is not called, nothing happens (no output).

```
In [ ]: # Example of a simple function
        def greet_bank_customer():
            print("Welcome to ABC Bank!")
```

```
In [ ]: # Call the function
        greet_bank_customer()
```

1.2 Arguments

Functions can accept **inputs**, known as **arguments** or **parameters**, which are passed into the function using the parentheses.

```
In [ ]: # Function to calculate simple interest
        def calculate_simple_interest(principal, rate, time):
            interest = principal * rate * time / 100
            print(f"The interest is: {interest}")
```

```
In [ ]: # Call the function with arguments
        calculate_simple_interest(1000, 5, 2)  # Principal = 1000, Rate = 5%, T
```

1.3 Return Values

Functions can **return** a value which can be assigned to an external variable. This obtains from the `return` statement.

```
In [ ]: # Function to calculate and return compound interest
        def calculate_compound_interest(principal, rate, time):
            amount = principal * (1 + rate/100)**time
            interest = amount - principal
            return interest
```

```
In [ ]: # Call the function and store the result
        compound_interest = calculate_compound_interest(1000, 5, 2)
        print("The compound interest is:", compound_interest)
```

1.4 Variable Scope

Variables defined inside a function are **local** to that function and cannot be accessed outside of it.

This is called the **scope of a variable**.

```
In [ ]: def calculate_balance():
            balance = 5000  # Local variable
            print("Balance inside the function:", balance)
```

```
In [ ]: calculate_balance()
```

```
In [ ]: print(balance)  # This will raise an error because balance is not acces
```

1.5 Built-in vs. User-defined Functions

Python provides many built-in functions like `print()`, `len()`, `sum()`, etc.

User-defined functions can also be used to extend and improve such functions.

In [ ]:

```
transactions = [100, -50, 200, -100]
total = sum(transactions)
print("Total balance after transactions:", total)
```

In [ ]:

```
def calculate_npv(cash_flows, discount_rate):
    npv = sum(cf / (1 + discount_rate) ** t for t, cf in enumerate(cash
    return npv

cash_flows = [-1000, 200, 300, 400, 500]
print("Net Present Value:", calculate_npv(cash_flows, 0.05))
```

2. Functional programming

Functional programming is a style of programming that treats computation as the evaluation of functions, just like in mathematics.

- Data is kept unchanged rather than modified
- Functions are written as **pure functions**: the same input always produces the same output, with no hidden effects
- **Higher-order** functions: can be passed to other functions or returned as results

Python supports functional programming features, making it easy to apply functional programming techniques.

> ***good practice***: *Avoiding loops as much as possible and making full use of list comprehensions and functional programming techniques.*

Math vs. Python Example

Mathematics:

$$f(x) = x^2$$

- Input $2 \rightarrow 4$
- Input $3 \rightarrow 9$

Python:

```python
def f(x):
    return x**2

f(2)  # 4
f(3)  # 9
```

2.1 Pure function

A **pure function** is a function that:

- Always produces the same output given the same input.
- Has no side effects (e.g., modifying external variables, changing mutable data, etc.).

Example of a Pure Function:

```
In [ ]: def add(a, b):
            return a + b
```

This function always returns the sum of `a` and `b` without modifying anything outside the function.

Non-Pure Function (with side effects):

In [ ]:
```
total = 0

def add_to_total(amount):
    global total
    total += amount
    return total
```

This function modifies the global variable `total`, which is considered a side effect and makes it a **non-pure function**.

2.2 Higher-order function

A **higher-order function** is a function that takes one or more functions as arguments or returns a function as its result.

Example of a Higher-Order Function:

In [ ]:
```python
def apply_twice(func, value):
    return func(func(value))

def double(x):
    return x * 2

print(apply_twice(double, 5))
```

2.3 Anonymous Functions (`lambda`)

In Python, **lambda functions** are **anonymous functions** that can have any number of arguments but only one expression.

`Lambdas` are useful for short functions without explicit definitions.

Lambda Function Syntax:

```
lambda arguments: expression
```

```
In [ ]: add = lambda a, b: a + b
        print(add(3, 4))  # Output: 7
```

```
In [ ]: # Lambda function for multiplication
        multiply = lambda x, y: x * y
        print(multiply(4, 2))  # Output: 8
```

```
In [ ]: # Lambda function with if-else to check if a number is even or odd
        even_or_odd = lambda x: 'even' if x % 2 == 0 else 'odd'
        print(even_or_odd(4))  # Output: even
        print(even_or_odd(7))  # Output: odd
```

In [ ]:

```
# Convert a string to uppercase using lambda
to_upper = lambda s: s.upper()
print(to_upper('hello'))  # Output: HELLO
```

```
In [ ]: # Lambda function to calculate the maximum of three numbers
        max_of_three = lambda a, b, c: a if (a > b and a > c) else (b if b > c
        print(max_of_three(10, 20, 15))  # Output: 20
```

2.4 Functional Programming Tools in Python

Python provides several built-in functions that are useful in functional programming

`map()`

Applies a given function to each item of an iterable (like a `list`) and returns a `map` object.

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

`filter()`

Filters items in an iterable based on a given function and returns an `iterator`.

```
In [ ]:
numbers = [1, 2, 3, 4, 5, 6]
evens_iterator = filter(lambda x: x % 2 == 0, numbers)
evens = list(evens_iterator)
print(evens)
print(evens_iterator)
```

Note: `iterators` in Python are exhausted after one use. Once converted to a list using `list(evens_iterator)`, the iterator is consumed and cannot be used again.

`reduce()`

Performs a rolling computation to sequential pairs of values in a `list`. It requires importing from the `functools` module.

In [ ]:
```python
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda a, b: a * b, numbers)
# Step 1: 1 * 2 = 2
# Step 2: 2 * 3 = 6
# Step 3: 6 * 4 = 24
print(result)
```

`zip()`

Combines two or more iterables into a single iterable of `tuples`.

In [ ]:
```python
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]

zipped = list(zip(names, scores))
print(zipped)
```

`sorted()`

Lambda functions can be used to define custom sorting behavior when using `sorted()`.

In [ ]:

```
# Sort a list of tuples based on the second element (using lambda)
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
sorted_pairs = sorted(pairs, key=lambda pair: pair[1])
print(sorted_pairs)
```

List Comprehension

Although list comprehension is not purely functional, it is closely aligned with functional programming ideas as it allows to create new lists in a concise way.

In [ ]:
```
squares = [x ** 2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

Dictionaries

Lambda functions can be used to get custom behavior when working with dictionaries, like finding the maximum or minimum key.

In [ ]:
```
# Find the key with the highest value in a dictionary
scores = {'Alice': 85, 'Bob': 90, 'Charlie': 88}
highest_scorer = max(scores, key=lambda k: scores[k])
print(highest_scorer)  # Output: Bob
```

Note: `max()` and `dict`:

1. `max(scores)`: Finds the maximum key (default behavior).

2. `max(scores.values())`: Finds the maximum value.

3. `max(scores, key=lambda k: scores[k])`: Finds the key corresponding to the maximum value.