Function

Kittikun Jitpairod

Introduction to Functions

Functions are reusable blocks of code that perform a specific task. They help us organize our code, make it more readable, and avoid repetition. Functions are a key aspect of the "Don't Repeat Yourself" (DRY) principle in programming.

In this session, we'll cover:

- Defining and calling functions
- Parameters and return values
- Variable scope
- Lambda functions
- Advanced function concepts

Defining Functions

Let's begin by creating a simple function:

```
def greet():
    print("Hello, World!")

# Calling the function
greet()
```

```
Hello, World!
```

Here's the anatomy of a function:

- The `def` keyword tells Python we're defining a function
- The function name (in this case, `greet`)
- Parentheses `()` (which can contain parameters, but are empty in this case)
- A colon `:` to start the function body
- The indented code block that makes up the function body

We call a function by using its name followed by parentheses.

Defining Functions with Input(s)

Let's create a function that takes a parameter(s):

```
def introduce(name, country):
    print(f"I'm, {name} from {country}.")

introduce("April", "Japan")
```

```
I'm, April from Japan.
```

The default value and suggested data type of input parameters can also be defined.

```
def introduce(name="Jonh",
    country:str="India"):

    print(f"I'm, {name} from {country}.")

introduce()
introduce("Bob")
introduce(country = 5) # use keyword argument
```

```
I'm, Jonh from India.
I'm, Bob from India.
I'm, Jonh from 5.
```

Defining Functions with Return Values

Functions return values by using `return` keyword:

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

result = add_numbers(5, 3)
print(f"5 + 3 = {result}")
```

```
5 + 3 = 8
```

Python functions can return multiple values:

```python
def calculate(a, b):
    add = a + b
    subtract = a - b
    return add, subtract

result1, result2 = calculate(10, 5)
print(f"10 + 5 = {result1}, 10 - 5 = {result2}")
```

```
10 + 5 = 15, 10 - 5 = 5
```

Variable Scope

Understanding variable scope is crucial when working with functions:

```python
x = 10  # Global variable

def print_x():
    print(x)  # Accessing global variable

print_x()

def change_x():
    x = 20  # Local variable
    print(x)

change_x()
print(x)  # Global x remains unchanged
```

```
10
20
10
```

If we want to modify a global variable inside a function, we use the `global` keyword:

```python
x = 10

def change_global_x():
    global x
    x = 20
    print(x)

change_global_x()
print(x)  # Global x is changed
```

```
20
20
```

Type Annotations

Type annotations allow us to specify the expected types of function parameters and return values.

Basic Type Annotations

Here's how we can add type annotations to a simple function:

```
def greet(name: str) -> str:
    return f"Hello, {name}!"

print(greet("Alice"))
```

```
Hello, Alice!
```

In this example, `: str` after `name` indicates that `name` should be a string, and `-> str` indicates that the function returns a string.

Type Checking

While Python itself doesn't enforce these types at runtime, we can use tools like mypy for static type checking:

```
def add_numbers(a: int, b: int) -> int:
    return a + b

result = add_numbers("5", "10")
print(result)
```

```
510
```

This code will run without errors in Python, but a type checker would flag it as an error because we're passing strings to a function that expects integers.