In the name of Allah

Hello Everybody

Let’s get started!

Functions

In Python, a function is a block of reusable and organized code that performs a specific task. Functions allow you to break down your program's logic into smaller, manageable pieces, making your code more modular, readable, and easier to maintain.

Function definition

Defining a function in Python involves specifying the function's name, its parameters (if any), and the code block that constitutes the function's body. Here's the basic syntax for defining a function in Python

```
# defining function
def function_name(parameters):
    # do somethings
    return result # optional
```

Function calling

Calling a function in Python involves using the function's name followed by parentheses that enclose the necessary arguments (if any). Here's how you call a function

```
# calling function
function_name(arguments)

# catching result
result = function_name(arguments)
```

Type of arguments

Positional Arguments

These are the most common type of arguments. They are matched with function parameters based on their position. The order of arguments matters, and they are assigned to parameters in the same order.

```python
def greet(name, age):
    print("Hello,", name)
    print("you are", age, "years old.")

greet("Alice", 25)
```

Keyword Arguments

In this type, arguments are matched with function parameters using their parameter names. This allows you to provide arguments in any order, which can enhance code readability.

```
def display_info(name, age):
    print("Name:", name)
    print("Age:", age)

display_info(age=30, name="John")
```

Default Arguments

Default arguments have predefined values in case an argument is not provided when calling the function. This is useful when some arguments are commonly used with specific values.

```python
def greet(name="Guest"):
    print("Hello,", name)

greet("Alice")   # Output: Hello, Alice
greet()          # Output: Hello, Guest
```

Variable-Length Argument Lists

Python allows functions to accept a variable number of arguments. There are two types:

1. Arbitrary Positional Arguments (*args): This allows a function to accept any number of positional arguments. These arguments are collected into a tuple.

```
def calculate_sum(*args):
    total = sum(args)
    return total

result = calculate_sum(2, 5, 7)
# Output: 14
```

Variable-Length Argument Lists

Python allows functions to accept a variable number of arguments. There are two types:

2. Arbitrary Keyword Arguments (**kwargs): This allows a function to accept any number of keyword arguments. These arguments are collected into a dictionary.

```
def display_properties(**kwargs):
    for key, value in kwargs.items():
        print(key, ":", value)

display_properties(color="red",
size="medium")
# Output:
# color : red
# size : medium
```

Combining Argument Types

You can combine different argument types in a single function.

```
def custom_greet(name, *titles, age=30,
**extra_info):
    print("Hello,", name)
    print("Age:", age)
    print("Titles:", titles)
    print("Extra Info:", extra_info)

custom_greet("Alice", "Ms.", "Doctor",
age=25, city="Wonderland")
```

Recursive functions

A recursive function is a function that calls itself as a part of its execution. In other words, it's a function that solves a problem by breaking it down into smaller instances of the same problem. Recursion is a powerful concept used to solve problems that can be broken down into simpler, similar subproblems.

Components

A recursive function typically has two components:

1. Base Case: This is the termination condition that prevents the function from calling itself indefinitely. It's the simplest case that can be solved directly without recursion.

2. Recursive Case: This is where the function calls itself with a modified version of the original problem, moving closer to the base case with each recursive call.

```python
def factorial(n):
    # Base case: factorial of 0 or 1 is 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial(n - 1)

result = factorial(5)
# Calculates 5! = 5 * 4 * 3 * 2 * 1 = 120
print(result)  # Output: 120
```

Lambda functions

In Python, a lambda function is a small, anonymous function that can have any number of arguments but only one expression. They are often used for short, simple operations where creating a full function using the def keyword might be overkill.

Defining lambda

Lambda functions are also known as "anonymous functions" because they don't need to be assigned a name. Instead, they are defined using the lambda keyword followed by the arguments and the expression.

```
lambda arguments: expression
```

Lambda Example

Here's a simple example of a lambda function that calculates the square of a number.

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

Lambda Trick!

Lambda functions are often used in places where a function is required as an argument, such as in the map(), filter(), and sorted() functions. Here's an example using map() to apply a lambda function to each element of a list

map Function

The map function applies a given function to each element of an iterable and returns an iterator that produces the transformed values.

```
map(function, iterable)

# Example
numbers = [1, 2, 3, 4, 5]
doubled_numbers = map(lambda x: x * 2,
numbers)
print(list(doubled_numbers))
# Output: [2, 4, 6, 8, 10]
```

filter Function

The filter function applies a given predicate function to each element of an iterable and returns an iterator that produces only the elements for which the predicate is True.

```
filter(predicate, iterable)

# Example
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0,
numbers)
print(list(even_numbers))
# Output: [2, 4]
```

sorted Function

The sorted function returns a new sorted list from the items in an iterable. It does not modify the original iterable.

```
sorted_list = sorted(iterable, key=None,
reverse=False)

# Example
numbers = [5, 2, 8, 1, 7]
sorted_numbers = sorted(numbers)
print(sorted_numbers)
# Output: [1, 2, 5, 7, 8]
```

sorted Function key

Key is an optional function that defines a custom sorting key for each item.

```
# Example 1
words = ["apple", "banana", "cherry",
"date"]
sorted_words = sorted(words, key=lambda
word: len(word))
print(sorted_words)
# Output: ['date', 'apple', 'banana',
'cherry']
```

sorted Function key

Key is an optional function that defines a custom sorting key for each item.

```
# Example 2
people = [
    {"name": "Alice", "age": 30},
    {"name": "Bob", "age": 25},
    {"name": "Charlie", "age": 35}
]

sorted_people = sorted(people, key=lambda
person: person["age"])
print(sorted_people)
# Output: [{'name': 'Bob', 'age': 25},
{'name': 'Alice', 'age': 30}, {'name':
'Charlie', 'age': 35}]
```

That’s all for today!