Notes on Python Functions

1. Introduction to Functions

A function is a block of reusable code that performs a specific task. Functions are a key feature in Python to achieve modular, organized, and maintainable code.

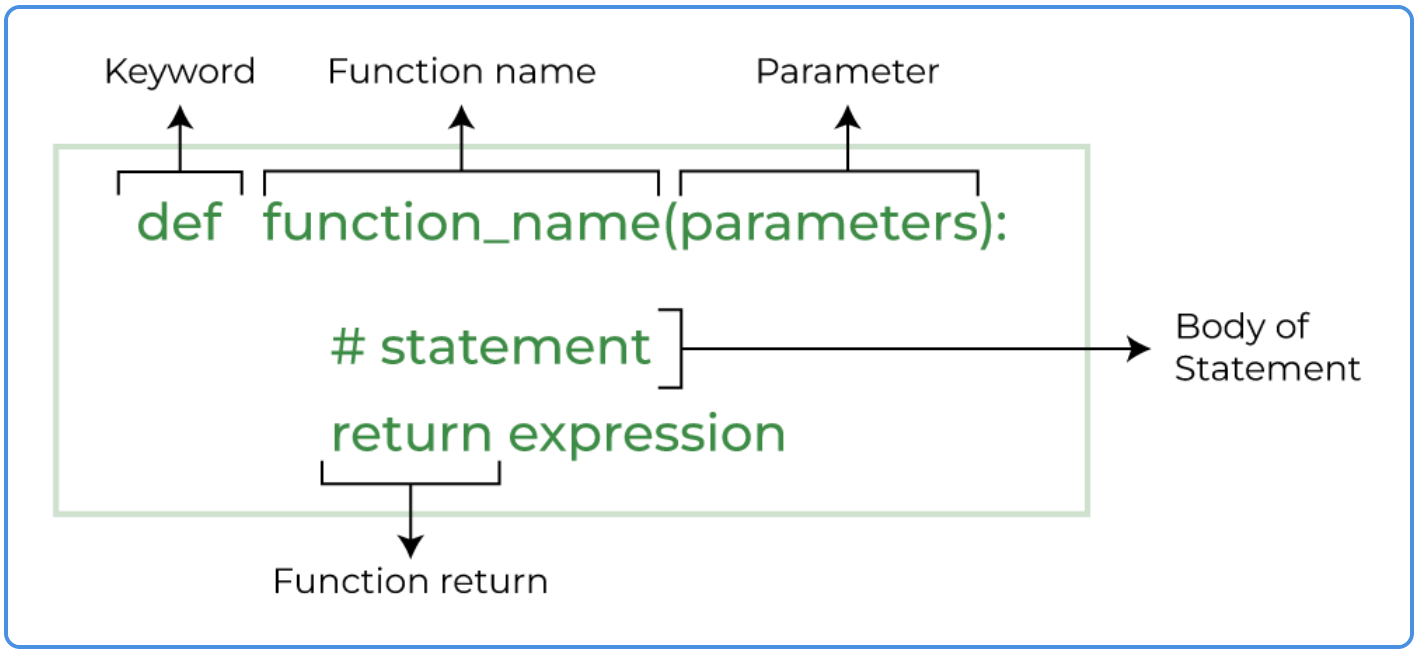
- **Code Reusability:** Once a function is defined, it can be used multiple times without rewriting the same code.
- **Improved Readability:** Breaking down a program into functions makes it easier to understand.
- **Easier Debugging and Maintenance:** Functions help isolate parts of the code, making it easier to spot issues.
- **Enables Modular Programming:** Functions allow complex programs to be broken down into simpler subprograms, which improves flexibility and collaboration.

2. Types of Functions

1. **Built-in Functions:** Python provides several built-in functions like `print()`, `len()`, `type()`, etc., which are ready to use.
2. **User-defined Functions:** These are functions created by the user to perform specific tasks not covered by built-in functions.

3. Defining and Calling a Function

Syntax:



Calling a Function:

```
function_name(arguments)
```

Example:

```
def greet(name):
    print(f"Hello, {name}!")
greet("Hemant") # Output: Hello, Hemant!
```

4. Function Components

- **Function Name:** A unique identifier for the function.
- **Parameters/Arguments:** Variables that accept input values for the function.
- **Docstring:** Optional text explaining what the function does.
- **Return Statement:** The value the function will send back to the caller.

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

```
result = add(5, 3)
print(result) # Output: 8
```

5. Function Parameters

1. **Positional Arguments:** Passed in the order they are defined.

```
def multiply(x, y):
    return x * y
print(multiply(2, 3)) # Output: 6
```

2. **Default Arguments:** Parameters with default values that are optional during the function call.

```
def greet(name="Student"):
    print(f"Hello, {name}!")
greet() # Output: Hello, Student!
```

3. **Keyword Arguments:** Specify the parameter names during the function call.

```
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")
display_info(age=20, name="Hemant")
```

4. **Variable-Length Arguments:**
   - ***args:** Accepts non-keyword variable-length arguments (tuple).

```
def sum_numbers(*args):
    return sum(args)
print(sum_numbers(1, 2, 3, 4)) # Output: 10
```

   - ****kwargs:** Accepts keyword variable-length arguments (dictionary).

```
def display_data(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
display_data(name="Hemant", age=20)
```

6. Scope of Variables

- **Local Scope:** Variables defined inside a function are local to that function.
- **Global Scope:** Variables defined outside of any function are accessible globally.
- **Global Keyword:** Allows modifying global variables inside a function.

```
x = 10
def modify_global():
    global x
    x = 20
modify_global()
print(x) # Output: 20
```

7. Anonymous Functions (Lambda)

Lambda functions are small, anonymous functions defined using the `lambda` keyword.

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

8. Higher-Order Functions

1. **Functions as Arguments:** Functions can be passed as arguments to other functions.

```
def apply_function(func, value):
    return func(value )
result = apply_function(lambda x: x**2, 5)
print(result) # Output: 25
```

2. **Functions as Return Values:** Functions can return other functions as results.

```
def outer_function():
    def inner_function():
        return "Hello from inner function"
    return inner_function
func = outer_function()
print(func()) # Output: Hello from inner function
```

9. Recursion

Recursion occurs when a function calls itself. It's useful for solving problems like factorial, Fibonacci sequence, etc.

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
print(factorial(5)) # Output: 120
```

10. Decorators

Definition: Functions that modify or extend the behavior of other functions.

Syntax:

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@decorator
def say_hello():
    print("Hello!")
say_hello()
```

Output:

```
Before function call
Hello!
After function call
```

11. Modules and Functions

Functions can be imported from external modules for use.

```
import math
print(math.sqrt(16)) # Output: 4.0
```

12. Best Practices

- **Meaningful Function Names:** Choose names that describe the function's purpose.
- **Keep Functions Small and Focused:** A function should do one thing and do it well.
- **Document Functions:** Use docstrings to describe what the function does and its parameters.
- **Use Default Arguments Judiciously:** Default arguments should have meaningful default values.
- **Avoid Side Effects:** Functions should avoid modifying global variables or causing unintended side effects.