Mastering Python

Functions

by Cahya Alkahfi

Table of contents

What is a function

A function is a separate code block that performs specific tasks. It allows us to group related code logically, making development, maintenance, and code understanding easier.

A function can take input arguments, perform specific operations, and return values. It is reusable, meaning you can call a function multiple times from different parts of a program without duplicating code.

Benefits of using functions:

- Break complex tasks into smaller, manageable parts
- Avoid redundant code by consolidating similar processes into functions
- Improve code readability by organizing it into functional blocks.
- Enhance code maintenance by dividing tasks into separate functions

There are many built-in functions in Python. Here are some examples:

```
# function to round a number to a spesific decimal place
round(3.141592, 4)   # output: 3.1416

# function to find the maximum value
max([5, 7, 8, 4])    # output: 8

# function to find the size of list
len(['a', 'b', 'c']) # output: 3

# function to sort a list
sorted([5, 7, 8, 4]) # output: [4, 5, 6, 8]
```

Creating a function

A function is created using the 'def' keyword followed by the function name and a pair of parentheses. If the function takes parameters, they are placed within the parentheses. A function can also have a return value, marked by the 'return' keyword followed by the value to be returned

```python
# simple function (just displaying text)
def greeting():
    print("Hello there!")

# function with 2 parameters
def print_info(name, location):
    print("Your info:")
    print(f">>> Name: {name}")
    print(f">>> Location: {location}")

# function with 3 parameters and a return value
def average(x1, x2, x3):
    avg = 1/3 * (x1 + x2 + x3)
    return avg
```

Calling a function

A function can be called by using the function name followed by a pair of parentheses. If the function has parameters, the function's arguments are placed within the parentheses. If the function has a return value, the function call can be assigned to a variable to store its return value.

```
# calling the `greeting`  function
greeting()
```

```
Hello there!
```

```
# calling the`print_info` function
# passing 2 values for `name` and `location`
print_info(name="Cahya Alkahfi", location="Jakarta")
```

```
Your info:
>>> Name: Cahya Alkahfi
>>> Location: Jakarta
```

```
# calling the `average` function
# storing the return value in `data_avg`
data_avg = average(x1=8, x2=6, x3=9)
print(data_avg)
```

```
7.666666666666666
```

Return Value

A function can return values of any type, ranging from simple types like int and string to more complex ones like lists, dictionaries, classes, and even other functions.

```python
def full_name(firstname, lastname):
    return f"{firstname} {lastname}"


my_full_name = full_name(firstname="CAHYA", lastname="ALKAHFI")

print(f"My name is {my_full_name}")
```

```
My name is CAHYA ALKAHFI
```

```python
def calculate_rectangle(length, width):
    area = length * width
    perim = 2 * (length + width)

    # return a dictionary
    return {"area": area,
            "perimeter": perim}


rect = calculate_rectangle(length=10, width=5)

print(f"Area of the rectangle is {rect['area']}")
print(f"Perimeter of the rectangle is {rect['perimeter']}")
```

```
Area of the rectangle is 50
Perimeter of the rectangle is 30
```

Parameter and Argument (1/3)

Parameter: A variable declared in the definition of a function. Parameters are used to receive values that will be used within the function. Parameters are placed within the parentheses when we define the function.

Argument: a concrete value provided to a function's parameter when calling the function. Arguments are placed inside the parentheses after the function name and are separated by commas if there is more than one.

```python
# function with 3 parameters
# (code, name, capital)
def country_info(code, name, capital):
    print("COUNTRY INFO:")
    print(f">>> Coutry Code : {code}")
    print(f">>> Name         : {name}")
    print(f">>> Capital      : {capital}")


# Calling the function with arguments
# "ID", "Indonesia", "DKI Jakarta"
country_info(code="ID", name="Indonesia", capital="DKI Jakarta")
```

```
COUNTRY INFO:
>>> Coutry Code : ID
>>> Name         : Indonesia
>>> Capital      : DKI Jakarta
```

more about parameter and argument >>>

Parameter and Argument (2/3)

Positional argument: We can omit parameter names as long as the order of the arguments matches the order of the parameters when the function is defined.

```
# Calling the function with arguments
# Without parameter names
# "ID", "Indonesia", "DKI Jakarta"
country_info("ID", "Indonesia", "DKI Jakarta")
```

On the other hand, we can provide arguments in any order as long as the parameter names are specified

```
# Calling the function with unordered arguments
country_info(capital="DKI Jakarta", code="ID", name="Indonesia")
```

both will produce the same result as before:

```
COUNTRY INFO:
>>> Coutry Code : ID
>>> Name         : Indonesia
>>> Capital      : DKI Jakarta
```

more about parameter and argument >>>

Parameter and Argument (3/3)

Default Parameter: we can provide default values to function parameters. When no argument is provided for that parameter, the default value will be used

```python
# exchange_rate has default value (15000)
def usd_to_idr(usd, exchange_rate=15_000):
    return usd*exchange_rate

# all arguments are provided
my_money = usd_to_idr(usd=100, exchange_rate=12_000)
print(f">>> My money is Rp.{my_money}")

# argument for `exchage_rate` is not provided
your_money = usd_to_idr(100)
print(f">>> Your money is Rp.{your_money}")

# all arguments are not provided
# (will raise TypeError)
# missing 1 required positional argument: 'usd'
his_money = usd_to_idr()
print(f">>> His money is Rp.{his_money}")
```

```
>>> My money is Rp.1200000
>>> Your money is Rp.1500000
TypeError: usd_to_idr() missing 1 required positional argument: 'usd'
```

*args

A special parameter used in a function definition to handle an unspecified number of arguments. This parameter allows us to pass in any number of arguments without specifying them individually in the function definition. The asterisk (*) preceding args is what makes it special, and we can use any different name but the asterisk must be present.

Arguments are Collected into a Tuple. We can access these arguments within the function using the tuple variable named args or any other variable we assign as the parameter for *args.

```python
def final_scores(math, *others):
  print(f"Math scores is {math}")
  print(f"Others scores are {others}")

  # `sum` and `len` are built-in function
  oth_avg = sum(others)/len(others)

  return 0.8*math + 0.2*oth_avg
```

```python
# calling the function
my_final_scores = final_scores(70, 100, 80, 90)
print(f"My final scores is {my_final_scores}")
```

```
Math scores is 70
Others scores are (100, 80, 90)
My final scores is 74.0
```

**kwargs

a special parameter used in a function definition to handle an unspecified number of keyword arguments. This parameter allows us to pass a variable number of keyword arguments to a function. The double asterisks (**) preceding "kwargs" is what makes it special, and like with *args, we can use a different name than "kwargs" but the double asterisks must be present. In this case, arguments are collected into a dictionary.

```
def final_scores(math, **others):
  print(f"Math scores is {math}")
  print(f"Others scores are {others}")

  oth_sum = 0
  # `others` is a dictionary
  for key, val in others.items():
    oth_sum +=val

  oth_avg = oth_sum/len(others)

  return 0.8*math + 0.2*oth_avg
```

```
# calling the function
my_final_scores = final_scores(70, sciences=100, english=80, art=90)
print(f"My final scores is {my_final_scores}")
```

```
Math scores is 70
Others scores are {'sciences': 100, 'english': 80, 'art': 90}
My final scores is 74.0
```

Combining any types of parameters

We can combine various types of parameters in a function. The rule is simple, all named parameters must come after all positional parameters

```python
def student_info(name, *args, **kwargs):
    print(f">>>>>>>>>> STUDENT INFO <<<<<<<<<<\n")
    print(f"- Name : {name}")
    print(f"- Test scores : {args}")    # args is a tuple
    print(f"- Other info:")
    for key, val in kwargs.items():    # kwargs is a dictionary
        print(f"    > {key}: {val}")
```

```python
# calling the function
student_info("John Doe",
             100, 80, 60, 80,
             sex='Male', age=18, status="Active")
```

```
>>>>>>>>>> STUDENT INFO <<<<<<<<<<

- Name : John Doe
- Test scores : (100, 80, 60, 80)
- Other info:
    > sex: Male
    > age: 18
    > status: Active
```

Variable Scope (1/2)

variable scope refers to the context in which a variable is defined and can be accessed. Python defines two main types of variable scope: global scope and local scope.

```
glob_var = 10

def my_function():
    print(glob_var)   # accessing the global variable


# glob-var is defined in the global scope
# can be accessed within the function my_function and outside of it.
my_function()
print(glob_var)
```

```
10
10
```

local scope variables can only be accessed from the function that declared it.

```
def other_function():
    loc_var = "sainsdata.id"
    print(loc_var)

other_function()

# Attempting to access loc_var outside the function will raise NameError.
```

```
sainsdata.id
```

Variable Scope (2/2)

Enclosing (Non-local) Scope: In some cases, we may have a variable defined in an enclosing function, and it is accessible to nested functions.

```
def outer_function():
    outer_variable = "sainsdata.id"

    def inner_function():
        # Accessing the variable from the enclosing function
        print(outer_variable)

    inner_function()


outer_function()
```

```
sainsdata.id
```

Global Keyword: If we need to modify a global variable from within a function, we can use the global keyword to indicate that the variable being modified is the one in the global scope.

```
glob_var = 10

def my_function():
    global glob_var  # using `global` keyword
    glob_var = 15

my_function()
print(glob_var)
```

```
15
```

Function as an argument (1/2)

In Python, functions are not just blocks of code that you call and execute; they can also be used as arguments to other functions. Functions that take other functions as arguments are commonly referred to as higher-order functions.

Many built-in function in python takes other function as an argument such as filter and map and many more.

```python
# function for checking whether a number is even or not
def is_even(number):
  return number % 2 == 0


# function for calculating cubic value from a number
def cubic(number):
  return number ** 3

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

# example of `filter` function
# 1st parameter is `function` that expect a function
# 2nd parameter is `iterable`
even_data = list(filter(is_even, data))
print(f"Even numbers: {even_data}")

# example of `map` function
# 1st parameter is `function` that expect a function
# 2nd parameter is `iterable`
cubic_data = list(map(cubic, data))
print(f"Cubic value of data: {cubic_data}")
```

```
Even numbers: [2, 4]
Cubic value of data: [1, 8, 27, 64, 125]
```

Function as an argument (2/2)

Another example:

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

def subtract(a, b):
  return a-b

# The `operation` function takes 3 arguments.
# `num1` and `num2` are expected to be numeric.
# `formula` is expected to be a function
def operation(num1, num2, formula):
  result = formula(num1, num2)
  print(f"{num1} {formula.__name__} {num2} = {result}")


operation(20, 5, add)
operation(20, 5, subtract)
```

```
20 add 5 = 25
20 subtract 5 = 15
```

Function as the return value

In Python, functions can also set as the return value for other function.

```python
# Function that returns another function
def get_operation(operator):
    if operator == '+':
        def add(x, y):
            return x + y
        return add
    elif operator == '-':
        def subtract(x, y):
            return x - y
        return subtract
    else:
        print("Invalid operator")

# Get the add function
add_func = get_operation('+')
result = add(20, 5)
print(result)  # Output: 25

# Get the subtract function
subtract_func = get_operation('-')
result = subtract_func(20, 5)
print(result)  # Output: 15
```

```
25
15
```

Lambda function (1/2)

Lambda function, also known as anonymous function, is a concise way to define small, one-line functions in Python. It is often used for short, simple operations and is particularly handy in situations where we need a quick, throwaway function.

```
lambda arguments: expression
```

- lambda: the keyword to define a lambda function,
- arguments: input parameters (if any) that the function will accept,
- expression: the expression that the function will perform. The result of this expression is implicitly returned.

```
add = lambda a,b : a + b
subtract = lambda a,b : a - b

a, b = 20, 5
a_plus_b = add(a, b)
a_minus_b = subtract(a, b)

print(f"{a} + {b} = {a_plus_b}")
print(f"{a} - {b} = {a_minus_b}")
```

```
20 + 5 = 25
20 - 5 = 15
```

Lambda function (2/2)

Let‘s redifine our function in the “Function as an argument section“. Instead of define the ` is_even` and `cubic` functions, we can directly pass lambda functions to `filter` and `map` functions.

```
data = [1, 2, 3, 4, 5]

# example of `filter` function
# 1st parameter is `function` that expect a function
# 2nd parameter is `iterable`
even_data = list(filter(lambda x: x % 2 == 0, data))
print(f"Even numbers: {even_data}")

# example of `map` function
# 1st parameter is `function` that expect a function
# 2nd parameter is `iterable`
cubic_data = list(map(lambda x: x ** 3, data))
print(f"Cubic value of data: {cubic_data}")
```

```
Even numbers: [2, 4]
Cubic value of data: [1, 8, 27, 64, 125]
```

Docstring (1/2)

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is used to provide documentation and information about the purpose and usage of the code. Docstrings are a valuable aspect of Python's self-documenting nature, aiding developers in understanding and using code effectively, especially when working with a large project.

Key Elements of a Docstring:

- Description: a docstring typically begins with a brief description of what the code does. This provides a high-level overview of the code's purpose.
- Parameters: If the code defines functions or methods that accept arguments, the docstring should detail these arguments, their types, and their purposes.
- Return Value: If a function or method returns a value, the docstring should specify the return type and describe the meaning of the returned value.
- Raises: If the code can raise exceptions, the docstring should mention the exceptions that may be raised and under what conditions.
- Examples: Including usage examples within the docstring helps users understand how to use the code effectively.

Docstring (2/2)

Example: factorial function with docstring

```python
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    This function takes a non-negative integer 'n' as input and returns its factorial.
    The factorial of a non-negative integer 'n' is the product of all positive integers
    from 1 to 'n'. The factorial of 0 is defined to be 1.

    Args:
        n (int): The non-negative integer for which the factorial is calculated.

    Returns:
        int: The factorial of 'n'.

    Raises:
        ValueError: If 'n' is a negative integer.

    Examples:
        >>> factorial(5)
        120
        >>> factorial(0)
        1
    """
    if n < 0:
        raise ValueError("Factorial is defined for non-negative integers only.")
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

THANK YOU

sainsdata.id