Lecture5

September 6, 2024

1 Lecture 5: Functions, Modules, and Packages

1.0.1 Objective:

"This lecture aims to introduce the concepts of `functions`, `modules`, and `packages` in Python. We will explore the role and creation of `functions` for organizing and reusing code, delve into `modules` for structuring code into manageable files, and examine `packages` for grouping related modules. By understanding these components, you'll gain the skills to build modular and maintainable Python programs." ****

The concepts of `functions`, `modules`, and `packages` are foundational to writing effective and maintainable Python code.

- **Functions** are vital for encapsulating and reusing blocks of code. They allow you to perform specific tasks and operations without repeating code, enhancing clarity and reducing errors.
- **Modules** help in organizing related functions, classes, and variables into separate files. This modular approach simplifies code management and improves readability by keeping related components together and making code easier to debug and update.
- **Packages** take modularity a step further by allowing you to group related modules into directories. This hierarchical structure supports the management of larger codebases, helps avoid name conflicts, and facilitates better dependency management.

These elements are crucial for writing organized, scalable, and efficient code, making development more manageable and collaboration more effective.

1.1 0. Functions

Simply defined, a function is a piece of code, a set of instructions organized to perform one or more well-defined tasks. In Python, functions are categorized into two types: built-in functions and user-defined functions.

- **Built-in functions** are functions that are directly integrated into Python's standard library.
- **User-defined functions** are written either by the current user or by other users.

1.1.1 0.0. Some built-in functions in Python

- **The `print()` function**: As we already know, the **print()** function displays the values of specified objects on the screen:

```
print("Hello", "everyone")
x = 12
```

```
print(x)
y = [1, "Monday", "12", 5, 3, "test value"]
print(y)
```

You can replace the default separator (a space) with another character (or even no character) using the `sep` argument:

```
print("Hello", "everyone", sep="****")
print("Hello", "everyone", sep="")
```

- **The `input()` function**: The **input()** function allows the user to enter a value for a given argument:

```
prenom = input("Enter your first name: ")
print("Hello,", prenom)
```

```
print("Please enter any positive number: ", end=" ")
ch = input()
num = int(ch)  # Convert the string to an integer
print("The square of", num, "is", num**2)
```

Note: It is important to note that the `input()` function always returns a string. If you need the user to enter a numeric value, you will have to convert the entered value (which will be of type string) into a numeric type using built-in functions like `int()` (for integers) or `float()` (for floating-point numbers).

1.1.2 0.1. User-defined functions

To define a function in Python, use the `def` keyword to declare the function name. The general syntax for defining a function is as follows:

```
def functionName([parameter1, parameter2, ..., parameterN]):

    """Documentation for the function."""

    <block_of_instructions>
```

In the function definition, the first string of characters (called a docstring) serves as documentation for the function, accessible via the interpreter using, for example, `help(functionName)`, or `functionName?` in *Jupyter*. It should be relevant, concise, and comprehensive. It may also include usage examples.

0.1.0. Definition of a simple function without arguments** The example below illustrates the definition of a simple function without arguments. The purpose of this function is to print the first 20 values of the multiplication table for 8.

```
def print_multiples_of_eight():
    """Prints the first 20 multiples of 8."""
    for i in range(1, 21):
        print(f"8 x {i} = {8 * i}")
```

```
# Calling the function
print_multiples_of_eight()
```

[1]:
```
def multiplication_table_8():
    """
    The purpose of this function is to display the first 20
    values of the multiplication table for 8.
    Input: None
    Output: The multiplication table for 8
    """
    n = 1
    while n <= 20:
        v = n * 8
        print(n, 'x', 8, '=', v, sep=' ')
        n = n + 1
```

To execute the function `multiplication_table_8()` that we just defined, simply reference it by its name as follows (anywhere in the main program):

```
multiplication_table_8()
```

[2]:
```
multiplication_table_8()  # Calls the function tableMultiplication8()
```

```
1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
4 x 8 = 32
5 x 8 = 40
6 x 8 = 48
7 x 8 = 56
8 x 8 = 64
9 x 8 = 72
10 x 8 = 80
11 x 8 = 88
12 x 8 = 96
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
18 x 8 = 144
19 x 8 = 152
20 x 8 = 160
```

[3]:
```
def print_multiples_of_eight():
    """Prints the first 20 multiples of 8."""
    for i in range(1, 21):
        print(f"8 x {i} = {8 * i}")
print_multiples_of_eight()
```

```
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
8 x 11 = 88
8 x 12 = 96
8 x 13 = 104
8 x 14 = 112
8 x 15 = 120
8 x 16 = 128
8 x 17 = 136
8 x 18 = 144
8 x 19 = 152
8 x 20 = 160
```

Exercise:

Propose a version of the function `tableMultiplication8()` using a `for` loop.
****

0.1.1. Définition d'une fonction dont les arguments sont des paramètres

A parameter is a variable that takes a constant value. In the previous example, we created a multiplication table for 8. We can generalize this function so that it returns the multiplication table for any specified number as an argument. Since these numbers are parameters, we need to define a function where the arguments are parameters. See the example below:

[4]:
```python
def tableMultiplication(base):
    n = 1
    while n <=20 :
        v=n*base
        print(n, 'x', base, '=', v, sep =' ')
        n = n +1
```

[5]:
```python
tableMultiplication(2) # returns the multiplication table for 2
print("============================\n")
tableMultiplication(8) # returns the multiplication table for 8
print("============================\n")
tableMultiplication(11) # returns the multiplication table for 11
```

```
1 x 2 = 2
2 x 2 = 4
3 x 2 = 6
```

```
4 x 2 = 8
5 x 2 = 10
6 x 2 = 12
7 x 2 = 14
8 x 2 = 16
9 x 2 = 18
10 x 2 = 20
11 x 2 = 22
12 x 2 = 24
13 x 2 = 26
14 x 2 = 28
15 x 2 = 30
16 x 2 = 32
17 x 2 = 34
18 x 2 = 36
19 x 2 = 38
20 x 2 = 40
============================

1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
4 x 8 = 32
5 x 8 = 40
6 x 8 = 48
7 x 8 = 56
8 x 8 = 64
9 x 8 = 72
10 x 8 = 80
11 x 8 = 88
12 x 8 = 96
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
18 x 8 = 144
19 x 8 = 152
20 x 8 = 160
============================

1 x 11 = 11
2 x 11 = 22
3 x 11 = 33
4 x 11 = 44
5 x 11 = 55
6 x 11 = 66
7 x 11 = 77
```

```
8 x 11 = 88
9 x 11 = 99
10 x 11 = 110
11 x 11 = 121
12 x 11 = 132
13 x 11 = 143
14 x 11 = 154
15 x 11 = 165
16 x 11 = 176
17 x 11 = 187
18 x 11 = 198
19 x 11 = 209
20 x 11 = 220
```

[6]:
```
# An example with a nicer display:


def multiplication_table(n):
    """
    The purpose of this function is to display the multiplication table for a␣
 ↪given number `n`.
    Input: n (int) - The number for which the multiplication table will be␣
 ↪generated.
    Output: The multiplication table for `n`.
    """
    print(f"Multiplication Table for {n}:")
    print("=" * 25)
    for i in range(1, 21):
        result = i * n
        print(f"{i:2} x {n:2} = {result:3}")
    print("=" * 25)

# Example usage:
multiplication_table(8)
```

```
Multiplication Table for 8:
=========================
 1 x  8 =   8
 2 x  8 =  16
 3 x  8 =  24
 4 x  8 =  32
 5 x  8 =  40
 6 x  8 =  48
 7 x  8 =  56
 8 x  8 =  64
 9 x  8 =  72
10 x  8 =  80
11 x  8 =  88
```

```
12 x  8 =  96
13 x  8 = 104
14 x  8 = 112
15 x  8 = 120
16 x  8 = 128
17 x  8 = 136
18 x  8 = 144
19 x  8 = 152
20 x  8 = 160
=========================
```

0.1.2. One or More Parameters, No Return

Example without the `return` statement, often referred to as a **procedure**. In this case, the function implicitly returns the value `None`:

```python
def table(base, start, end):
    """Displays the multiplication table of <base> from <start> to <end>."""
    n = start
    while n <= end:
        print(n, 'x', base, '=', n * base, end=" ")
        n += 1

# Example call:
table(7, 2, 11)
# 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42
# 7 x 7 = 49 8 x 7 = 56 9 x 7 = 63 10 x 7 = 70 11 x 7 = 77
```

---

[7]:
```python
def table(base, start, end):
    """Displays the multiplication table of <base> from <start> to <end>."""
    n = start
    while n <= end:
        print(n, 'x', base, '=', n * base, end=" ")
        n += 1
table(7, 2, 11)
```

```
2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49 8 x 7 = 56 9 x
7 = 63 10 x 7 = 70 11 x 7 = 77
```

[8]:
```python
# Cool display

def table(base, start, end):
    """Displays the multiplication table of <base> from <start> to <end>."""
    print(f"Multiplication Table for {base}:")
    print(f"{'Number':<10}{'Result':<10}")
    print("-" * 20)
    for n in range(start, end + 1):
        print(f"{n:<10}{n * base:<10}")
```

```
# Example call:
table(7, 2, 11)
```

```
Multiplication Table for 7:
Number    Result
--------------------
2         14
3         21
4         28
5         35
6         42
7         49
8         56
9         63
10        70
11        77
```

One or more parameters, use of Return

- Example with a single `return`:

```
def square(x):
    """
    Calculate the square of a number.

    Args:
    x (float): The number to be squared.

    Returns:
    float: The square of the input number.
    """
    return x**2

def squareArea(r):
    """
    Calculate the area of a square given the length of its side.

    Args:
    r (float): The length of the side of the square.

    Returns:
    float: The area of the square.
    """
    return square(r)

# Input for the side length and display of the area
side = float(input('Side: '))
print("Square area =", squareArea(side))
```

[9]:
```python
def square(x):
    """
    Calculate the square of a number.

    Args:
    x (float): The number to be squared.

    Returns:
    float: The square of the input number.
    """
    return x**2

def squareArea(r):
    """
    Calculate the area of a square given the length of its side.

    Args:
    r (float): The length of the side of the square.

    Returns:
    float: The area of the square.
    """
    return square(r)

# Input for the side length and display of the area
side = float(input('Side: '))
print("Square area =", squareArea(side))
```

```
Side: 76
Square area = 5776.0
```

- Example with multiple returns:

```python
PI = 3.14

def surfaceVolumeSphere(r):
    """
    Calculate the surface area and volume of a sphere.

    Args:
    r (float): The radius of the sphere.

    Returns:
    tuple: A tuple containing the surface area and volume of the sphere.
    """
    surf = 4.0 * PI * r**2
    vol = surf * r / 3
    return surf, vol
```

```
# Main program
radius = float(input('Radius: '))
s, v = surfaceVolumeSphere(radius)
print("Sphere with surface {:g} and volume {:g}".format(s, v))
```

[10]:
```
PI = 3.14

def surfaceVolumeSphere(r):
    """
    Calculate the surface area and volume of a sphere.

    Args:
    r (float): The radius of the sphere.

    Returns:
    tuple: A tuple containing the surface area and volume of the sphere.
    """
    surf = 4.0 * PI * r**2
    vol = surf * r / 3
    return surf, vol

radius = float(input('Radius: '))
s, v = surfaceVolumeSphere(radius)
print("Sphere with surface {:g} and volume {:g}".format(s, v))
```

```
Radius: 87
Sphere with surface 95066.6 and volume 2.75693e+06
```

1.1.3 0.2. Passing a Function as a Parameter

```
def tabulate(function, lowerBound, upperBound, numSteps):
    """Display the values of <function>. Conditions: (lowerBound < upperBound) and (numSteps >
    h, x = (upperBound - lowerBound) / float(numSteps), lowerBound
    while x <= upperBound:
        y = function(x)
        print("f({:.2f}) = {:.2f}".format(x, y))
        x += h

def myFunction(x):
    return 2 * x**3 + x - 5

tabulate(myFunction, -5, 5, 10)
# f(-5.00) = -260.00
# f(-4.00) = -137.00
# ...
# f(5.00) = 250.00
```

[11]:
```
def tabulate(function, lowerBound, upperBound, numSteps):
    """Display the values of <function>. Conditions: (lowerBound < upperBound)␣
↪and (numSteps > 0)"""
    h, x = (upperBound - lowerBound) / float(numSteps), lowerBound
    while x <= upperBound:
        y = function(x)
        print("f({:.2f}) = {:.2f}".format(x, y))
        x += h


def myFunction(x):
    return 2 * x**3 + x - 5


tabulate(myFunction, -5, 5, 10)
```

```
f(-5.00) = -260.00
f(-4.00) = -137.00
f(-3.00) = -62.00
f(-2.00) = -23.00
f(-1.00) = -8.00
f(0.00) = -5.00
f(1.00) = -2.00
f(2.00) = 13.00
f(3.00) = 52.00
f(4.00) = 127.00
f(5.00) = 250.00
```

1.1.4 Improved Display and Explanation

Here's the improved version of the code with a more structured and readable output, followed by an explanation:

```
def tabulate(function, lower_bound, upper_bound, num_steps):
    """
    Displays the values of <function> within the range from <lower_bound> to <upper_bound>.
    The function is evaluated at evenly spaced points determined by <num_steps>.

    Parameters:
    function (callable): The function to be tabulated.
    lower_bound (float): The starting point of the range.
    upper_bound (float): The ending point of the range.
    num_steps (int): The number of intervals in the range.

    Conditions:
    - lower_bound < upper_bound
    - num_steps > 0
    """
    step_size = (upper_bound - lower_bound) / float(num_steps)
    x = lower_bound

    print("Tabulation of the function from {:.2f} to {:.2f} with {} steps:".format(lower_bound
    print("----------------------------------------------------------")

    while x <= upper_bound:
        y = function(x)
        print("f({:+.2f}) = {:+.2f}".format(x, y))
        x += step_size

def my_function(x):
    """A sample function: f(x) = 2x^3 + x - 5"""
    return 2 * x**3 + x - 5

# Tabulate the function my_function from -5 to 5 with 10 steps
tabulate(my_function, -5, 5, 10)
```

1.1.5 Example Output:

```
Tabulation of the function from -5.00 to 5.00 with 10 steps:
----------------------------------------------------------
f(-5.00) = -260.00
f(-4.00) = -137.00
f(-3.00) = -68.00
f(-2.00) = -27.00
f(-1.00) = -8.00
f(+0.00) = -5.00
f(+1.00) = -2.00
f(+2.00) = +19.00
f(+3.00) = +82.00
f(+4.00) = +203.00
f(+5.00) = +395.00
```

1.1.6 Explanation:

1. **Function Parameters:**
   - **`function`**: This parameter takes a function as its value. In the example, `my_function` is passed, which defines the mathematical expression (f(x) = 2x^3 + x - 5).
   - **`lower_bound`**: The starting point of the range where the function will be evaluated.
   - **`upper_bound`**: The endpoint of the range.
   - **`num_steps`**: The number of intervals between the lower and upper bounds, determining how many times the function will be evaluated.
2. **Step Size Calculation:**
   - The **`step_size`** is calculated by dividing the difference between `upper_bound` and `lower_bound` by the number of steps (`num_steps`). This determines how much `x` will increment on each loop iteration.
3. **Looping and Evaluation:**
   - A `while` loop iterates from `lower_bound` to `upper_bound`, calculating the function value

    y at each point x.
  - The loop prints each evaluated pair (x, y) in a well-formatted string, showing the results of the function evaluation.
4. **Output Formatting:**
   - The output is formatted to display the values with two decimal places for clarity and readability. The plus (+) sign before the numbers ensures both positive and negative signs are displayed.

This structure makes it easy to understand how the function behaves over a specified range, providing clear and organized output.

[12]:
```python
def tabulate(function, lower_bound, upper_bound, num_steps):
    """
    Displays the values of <function> within the range from <lower_bound> to␣
 ↪<upper_bound>.
    The function is evaluated at evenly spaced points determined by <num_steps>.

    Parameters:
    function (callable): The function to be tabulated.
    lower_bound (float): The starting point of the range.
    upper_bound (float): The ending point of the range.
    num_steps (int): The number of intervals in the range.

    Conditions:
    - lower_bound < upper_bound
    - num_steps > 0
    """
    step_size = (upper_bound - lower_bound) / float(num_steps)
    x = lower_bound

    print("Tabulation of the function from {:.2f} to {:.2f} with {} steps:".
 ↪format(lower_bound, upper_bound, num_steps))
    print("----------------------------------------------------------")

    while x <= upper_bound:
        y = function(x)
        print("f({:+.2f}) = {:+.2f}".format(x, y))
        x += step_size

def my_function(x):
    """A sample function: f(x) = 2x^3 + x - 5"""
    return 2 * x**3 + x - 5

# Tabulate the function my_function from -5 to 5 with 10 steps
tabulate(my_function, -5, 5, 10)
```

```
Tabulation of the function from -5.00 to 5.00 with 10 steps:
----------------------------------------------------------
```

```
f(-5.00) = -260.00
f(-4.00) = -137.00
f(-3.00) = -62.00
f(-2.00) = -23.00
f(-1.00) = -8.00
f(+0.00) = -5.00
f(+1.00) = -2.00
f(+2.00) = +13.00
f(+3.00) = +52.00
f(+4.00) = +127.00
f(+5.00) = +250.00
```

1.1.7 0.3. Defining default values for function arguments**

When defining a function, it is often recommended to set default values for certain arguments, especially optional ones. By defining default values for a function's arguments, it becomes possible to call the function with only some of the expected arguments. Here are some examples:

```
def greeting(name, title='Monsieur'):
    """
    Displays a greeting message with the given name and an optional title.

    Parameters:
    name (str): The name of the person to greet.
    title (str, optional): The title of the person (default is 'Monsieur').

    Returns:
    None
    """
    print("Bonjour", title, name)
```

- **Explanation**:
  The `greeting` function has two arguments: `name` and `title`. A default value ('Monsieur') has been set for the `title` argument. Therefore, when the `greeting` function is called with only the `name` argument (omitting the `title` argument), the function will use the default value 'Monsieur'.

[13]:
```
def greeting(name, title='Monsieur'):
    """
    Displays a greeting message with the given name and an optional title.

    Parameters:
    name (str): The name of the person to greet.
    title (str, optional): The title of the person (default is 'Monsieur').

    Returns:
    None
    """
    print("Bonjour", title, name)
```

Example Usage:

```
greeting('Dupont')
# Output: Bonjour Monsieur Dupont
```

[14]:
```
greeting('Dupont')
# Output: Bonjour Monsieur Dupont
```

```
Bonjour Monsieur Dupont
```

- **Explanation**:
  When the function is called with both arguments, the default value is overridden by the provided value.

Example:

```
greeting('Dupont', 'Mademoiselle')
# Output: Bonjour Mademoiselle Dupont
```

[15]:
```
greeting('Dupont', 'Mademoiselle')
# Output: Bonjour Mademoiselle Dupont
```

```
Bonjour Mademoiselle Dupont
```

- **Explanation**:
  By defining default values for a function's arguments, you can make the function calls more flexible, allowing it to be called with only a subset of the expected arguments when needed.

Note:

Arguments without default values must be specified before arguments with default values. If this rule is not followed, Python will raise an error during execution. For example, the following function definition is incorrect:

```
def greeting(title='Monsieur', name):
    """
    This function is incorrectly defined and will raise an error because the argument
    with a default value ('title') comes before an argument without a default value ('name').

    Parameters:
    title (str, optional): The title of the person (default is 'Monsieur').
    name (str): The name of the person to greet.

    Returns:
    None
    """
```

- **Explanation**:
  The `greeting` function is defined incorrectly because `title`, an argument with a default value, is placed before `name`, an argument without a default value. In Python, arguments without default values must precede those with default values to avoid a syntax error.

[16]:
```
def salutation(titre='Monsieur', name):
  Cell In [16], line 1
    def salutation(titre='Monsieur', name):
                                     ^
SyntaxError: non-default argument follows default argument
```

1.1.8 0.4. Lambda Functions

A lambda function is an anonymous function, meaning it is a function that consists of a block of instructions that can be called and reused like a regular function but without a name. Lambda functions are typically used for very short functions with few instructions, which do not require a full function definition using the `def` keyword.

The general syntax for defining a lambda function is as follows:

```
lambda arg1, arg2, ..., argN : instruction_block (or formula)
```

The example below illustrates the definition of a lambda function:

```
lambda x, y : x * y
```

Notons toutefois que même si la fonction lambda n'est pas définie avec un nom, pour récupérer la valeur renvoyée, lors de l'appel de la fonction, il faut l'assigner à une nouvelle variable. L'exemple ci-dessous illustre l'appel de la fonction lambda précédente en prenant x=2 et y=3.

```
x = lambda x, y : x * y
x(2,3)
```

```
[17]: x = lambda x, y : x * y
      x(2,3)
```

```
[17]: 6
```

Note: When we define variables inside a function, these variables are only accessible within that function itself. These variables are referred to as « `local` » to the function. However, when variables are defined outside of the function in the main program body, they are called « `global` » variables. The content of a global variable is visible and accessible from within a function, but the function cannot modify the value of the variable.

```
def myFunction():
    p = 20
    print(p, q)

p = 15
q = 38

print(p, q)  # Outputs: 15 38
myFunction()  # Calls the function, Outputs: 20 38
print(p, q)  # Outputs: 15 38
```

[18]:
```python
def myFunction():
    p = 20
    print(p, q)

p = 15
q = 38

print(p, q)  # Outputs: 15 38
myFunction()  # Calls the function, Outputs: 20 38
print(p, q)  # Outputs: 15 38
```

```
15 38
20 38
15 38
```

Explanation:

1. **Global and Local Variables:**
   - The variables `p` and `q` are defined in the global scope, outside the function `myFunction()`. Hence, they are global variables.
   - Inside `myFunction()`, a new variable `p` is defined with the value `20`. This `p` is local to the function and does not affect the global `p`.
2. **Output Analysis:**
   - **Before Function Call (`print(p, q)`):** This prints the global values of `p` and `q`, which are `15` and `38`, respectively.
   - **Inside Function (`myFunction()`):** When `myFunction()` is called, it prints the local `p` (which is `20`) and the global `q` (which remains `38`). The local `p` inside the function shadows the global `p` but does not change it.
   - **After Function Call (`print(p, q)`):** This again prints the global values of `p` and `q`, which are still `15` and `38`, as the local `p` inside the function did not alter the global `p`.

However, you can modify this default behavior by allowing the function to modify the value of a global variable. To do this, you must explicitly declare the variable as global within the function. Example:

```python
def myFunction():
    global p
    p = 20
    print(p, q)

p = 15
q = 38

print(p, q)  # Outputs: 15 38
myFunction()  # Calls the function, Outputs: 20 38
print(p, q)  # Outputs: 20 38
```

Explanation:

1. **Global Declaration:**

- Inside `myFunction()`, the `global` keyword is used to indicate that `p` refers to the global variable `p`, not a new local variable.

2. **Output Analysis:**
   - **Before Function Call (`print(p, q)`):** This prints the global values of `p` and `q`, which are 15 and 38, respectively.
   - **Inside Function (`myFunction()`):** When `myFunction()` is called, it sets the global `p` to 20 and prints this new value of `p` along with the global `q` (which remains 38).
   - **After Function Call (`print(p, q)`):** This prints the updated global values of `p` and `q`, which are now 20 and 38, respectively, reflecting the change made inside `myFunction()`.

[19]:
```python
def myFunction():
    global p
    p = 20
    print(p, q)

p = 15
q = 38

print(p, q)  # Outputs: 15 38
myFunction()  # Calls the function, Outputs: 20 38
print(p, q)  # Outputs: 20 38
```

```
15 38
20 38
20 38
```

1.1.9 0.5. Arbitrary number of arguments

0.5.0. Passing a Tuple

[20]:
```python
def somme(*args):
    """Returns the sum of the <tuple>."""
    resultat = 0
    for nombre in args:
        resultat += nombre
    return resultat

# Example calls:
print(somme(23)) # 23
print(somme(23, 42, 13)) # 78
```

```
23
78
```

~~~
If the function has multiple arguments, the *tuple* is in the last position. It is also possib

```python
def somme(a, b, c):
    return a + b + c
~~~

```
# Example call:
elements = (2, 4, 6)
print(somme(*elements)) # 12
```

0.5.1. Passing a Dictionary

[21]:
```
def unDict(**kargs):
    return kargs

# Examples of calls
## Using named parameters:
print(unDict(a=23, b=42)) # {'a': 23, 'b': 42}

## Providing a dictionary:
mots = {'d': 85, 'e': 14, 'f': 9}
print(unDict(**mots)) # {'d': 85, 'e': 14, 'f': 9}
```

```
{'a': 23, 'b': 42}
{'d': 85, 'e': 14, 'f': 9}
```

1.1.10 0.6. Documenting a Function

After creating a function (especially a relatively long and complex one), it is highly recommended to document it to allow other users to understand it quickly. Function documentation is typically a string that provides an overview of the function and useful details. This description is generally specified right after the function's name declaration and before the definition of other instruction blocks. The example below illustrates how to document a function and how to access this documentation when needed.

[22]:
```
def volumeSphere():
    """ This program calculates the volume of a sphere.
    The function is defined with a single required argument r
    which represents the radius of the sphere.
    It can take any positive value."""
    r = float(input("Enter the radius of the sphere: "))
    PI = 3.14
    return (4 * PI * r**3) / 3
```

In the definition of the `volumeSphere` function, the string does not play any functional role in the script; it is treated by Python as a simple comment but is stored as internal documentation for the function. This documentation is stored in an attribute called `__doc__`. To display this attribute, you use:

```
print(volumeSphere.__doc__)
```

[23]:
```
print(volumeSphere.__doc__)
```

```
 This program calculates the volume of a sphere.
    The function is defined with a single required argument r
    which represents the radius of the sphere.
    It can take any positive value.
```

1.2 1. Modules

A Python program is generally composed of several source files, called *modules*. Their names have the `.py` suffix. If correctly coded, modules should be independent of each other and reusable on demand in other programs.

Modules are files that group sets of functions. A **module** is an independent file that allows a program to be split into several scripts. This mechanism allows for the efficient creation of function or class libraries.

Advantages of modules: - Code reuse; - Documentation and tests can be integrated into the module; - Implementation of shared services or data; - Partitioning of the system's namespace.

Just as dictionaries are collections of objects (lists, tuples, sets, etc.), modules are collections of functions that perform related tasks. For example, the `math` module contains a number of mathematical functions such as sine, cosine, tangent, square root, etc. Many modules are already pre-installed in Python's standard library. However, to perform certain specific tasks, you often need to install additional modules (e.g., `numpy`, `scipy`, `matplotlib`, `pandas`, etc.).

1.2.1 1.0. Importing a Module

There are two possible syntaxes:

- The `import nom_module` command imports all objects from the module:

  ```
  import tkinter
  ```

- The `from <nom_module> import obj1, obj2` command imports only the specified objects `obj1, obj2...` from the module:

  ```
  from math import pi, sin, log
  ```

It is recommended to import in the following order: - Standard library modules; - Third-party library modules; - Personal modules.

1.2.2 1.1. The Standard Library

It is often said that Python comes "batteries included" due to its standard library, which is rich with over 200 packages and modules designed to address a wide range of common problems. See The Python Standard Library.

[24]:
```
import math
dir(math)  # To see the list of functions and attributes in the module.
```

[24]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
```

```
'atan',
'atan2',
'atanh',
'cbrt',
'ceil',
'comb',
'copysign',
'cos',
'cosh',
'degrees',
'dist',
'e',
'erf',
'erfc',
'exp',
'exp2',
'expm1',
'fabs',
'factorial',
'floor',
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'hypot',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'isqrt',
'lcm',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'modf',
'nan',
'nextafter',
'perm',
'pi',
'pow',
'prod',
'radians',
'remainder',
'sin',
'sinh',
'sqrt',
'tan',
'tanh',
'tau',
'trunc',
'ulp']
```

[25]:
```
help(math.gamma)  # Displays the documentation for the gamma function in the␣
 ↪math module.
```

```
Help on built-in function gamma in module math:

gamma(x, /)
    Gamma function at x.
```

[26]:
```
from math import sin  # Imports the sine function

from math import cos, sin, tan, pi  # Imports the cosine, sine, tangent␣
 ↪functions, and the value of pi (3.14)
```

[27]:
```
from math import *  # Imports all functions and constants from the math module␣
 ↪(equivalent to import math)
```

Some uses of the math function:

```
from math import *

v = 16  # defines a variable v
x = sqrt(v)  # Returns the square root of v
y = exp(v)   # Returns the exponential of v
z = log(v)   # Returns the natural logarithm of v
```

[28]:
```
from math import *
v = 16  # defines a variable v
x = sqrt(v)  # Returns the square root of v
y = exp(v)   # Returns the exponential of v
z = log(v)   # Returns the natural logarithm of v
print(v, x, y, z)
```

```
16 4.0 8886110.520507872 2.772588722239781
```

Some examples of using the random module:

```
from random import random, randint, seed, uniform, randrange, sample, shuffle  # Imports some
```

[29]:
```
import random

x = random.random()  # Returns a random number between 0.0 and 1.0
print(x)
```

```
0.19866208294418142
```

[30]:
```
import random

x = random.randint(5, 17)  # Returns a random integer between 5 and 17␣
  ↪(inclusive)
print(x)
```

```
9
```

[31]:
```
import random

x = random.uniform(5, 17)  # Returns a random floating-point number between 5␣
  ↪and 17
print(x)
```

```
14.1213978301999
```

1.3 Feel free to explore the `turtle`, `time`, `decimal`, `fractions`, `cmath` modules.

1.3.1 1.2. Third-Party Libraries

In addition to the modules included in the standard Python distribution, you can find libraries in various fields: - Scientific - Databases - Functional testing and quality control - 3D - …

The PYPI (The Python Package Index) lists thousands of modules and packages!

1.3.2 1.3. Define and use your own module

You can create your own module by gathering several functions into a single script and saving it with the `.py` extension in the current directory. The name should be simple and not create ambiguity with other Python objects. For example, you might choose `myprogram.py`.

Once the script is saved in the current directory, you can import the module like a standard module, and all its functions (and variables) become accessible. The module is imported using the command:

`import myprogram`

You can then use the functions from the module as you would with any standard module. For example:

```
# myprogram.py
def greet(name):
    """Returns a greeting message."""
    return f"Hello, {name}!"

def add(a, b):
    """Returns the sum of two numbers."""
    return a + b

# main.py
import myprogram

print(myprogram.greet("Alice"))  # Output: Hello, Alice!
print(myprogram.add(5, 7))       # Output: 12
```

Open your preferred text editor and write the following code, which you should save as `cube_m.py`:

```
# A module named cube_m.py
def cube(y):
    """Calculates the cube of the parameter <y>."""
    return y**3


# Self-test -----------------------------------------------------
if __name__ == "__main__": # False when imported ==> ignored
    help(cube)
    # displays the docstring of the function
    print("cube of 9:", cube(9)) # cube of 9: 729
```

```
# Using this module. We import the function cube() from the file cube_m.py:
from cube_m import cube

for i in range(1, 4):
    print("cube of", i, "=", cube(i), end=" ")
# cube of 1 = 1 cube of 2 = 8 cube of 3 = 27
```

1.4 2. Package

A second level of organization allows for structuring the code: Python files can be organized in a directory hierarchy called a `package`.

More simply, a *package* is a module containing other modules. The modules in a package can be *sub-packages*, creating a tree-like structure. In summary, a package is simply a directory that contains modules and an `__init__.py` file describing the package's structure. Example:

1.4.1 In a terminal, do the following:

- `mkdir monpackage`;
- `cd monpackage`;
- `touch __init__.py`;
- `touch mesfonctions.py`
  - This file contains two Python functions:

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

def soustraire(a, b):
    return a - b
```

- touch mesattributs.py
  - This file contains two constants:
    ```
    x = 100
    y = 95
    ```

1.4.2 Then do this:

This sequence of code demonstrates how to use the functions and constants defined in the monpackage package:

1. **Importing and Using Functions:**

   ```
   from monpackage import mesfonctions
   mesfonctions.additionner(23, 89) == 112  # Returns True
   ```

   - Here, the additionner function is imported from the mesfonctions module within the monpackage package.
   - mesfonctions.additionner(23, 89) calls the additionner function with arguments 23 and 89.
   - The function returns 112, which matches the expected result, so the expression mesfonctions.additionner(23, 89) == 112 evaluates to True.

2. **Importing and Using Constants:**

   ```
   from monpackage import mesattributs
   mesattributs.x == 100  # Returns True
   ```

   - Here, the constant x is imported from the mesattributs module within the monpackage package.
   - mesattributs.x accesses the constant x which is defined as 100.
   - The expression mesattributs.x == 100 evaluates to True since the value of x is indeed 100.

[17]:
```
from monpackage import mesfonctions
```

[18]:
```
mesfonctions.additionner(23,89) == 112  # Returns True
```

[19]:
```
from monpackage import mesattributs
```

[20]:
```
mesattributs.x == 100 # Returns True
```

1.5 Further reading

0. Programming with Python; Section 22
1. Programming with Python; Section 23
2. Programming with Python; Section 25