Chapter 4: Functions

4.1 Introduction to Functions

User-Defined Functions / Arguments and Parameters

What is a function?

A function is a named block of code that performs a specific task. It helps you avoid repeating code, and makes programs more modular and easier to understand. For example: “calculate the square of a number”, “print a greeting”, etc.

How to define and call a user-defined function:

In Python, you use the def keyword, then function name, parentheses (with parameters if any), a colon, then the indented body. Example:

```
def greet(name):              # name is a parameter
  print("Hello, " + name + "!")

greet("Alice")              # calling the function, passing “Alice” as argument
```

Here:

- greet is the function name.
- name is a parameter (a placeholder inside the function).
- When we call greet("Alice"), the value "Alice" is the argument.
- Inside the function, name refers to "Alice" during that call.

Why functions?

- They allow reuse: you write once, call many times.
- They make code modular: each function does one thing.
- They separate concerns: you can test, debug, maintain functions independently.

More on arguments and parameters:

- Parameter = the variable name in the function definition.
- Argument = the actual value you pass when calling the function.
- You can have zero parameters, one, or many.
- Example of multiple parameters:

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

result = add(3, 5)    # arguments are 3 and 5
print(result)         # prints 8
```

Return values:

Functions may optionally return values using the return statement. If no return is given, the function returns None by default. Example:

```
def square(x):
  return x * x

y = square(4)         # y gets the value 16
```

Important rules / good practices:

- Choose a meaningful function name (describe what it does).
- Write parameters that make sense.
- Avoid doing too many unrelated things in one function (single responsibility).
- Use docstrings (optional) to explain the function’s purpose.

4.2 Scope of a Variable

Global Variable / Local Variable

What is scope?

Scope refers to the region of the program where a variable is accessible/valid. In Python, variables defined inside a function are **local** to that function; variables defined outside functions (in the main body) are **global**.

Local variables:

- Defined inside a function.
- Only accessible inside that function (or block) where they are defined.
- Example:

```
def foo():
    x = 10       # x is local to foo
    print(x)

foo()
# print(x)      # This would cause an error: x is not defined here
```

Global variables:

- Defined at the top level of a module (outside any function).
- Accessible by functions (unless shadowed by a local variable with same name).
- Example:

```
count = 0       # global variable

def increment():
    global count   # tell Python we mean the global one
    count += 1

increment()
print(count)      # prints 1
```

Using global keyword:

When inside a function you want to modify a global variable, you must declare it global. Otherwise, assignment inside the function will create a new local variable.

Why understand scope?

- Avoid unexpected behaviour (e.g., accidentally modifying global when you meant new local).
- Prevent name collisions.
- Help readability: knowing which variables affect what parts of code.

4.3 Python Standard Library

Built-in functions / Input or Output / Mathematical Functions / Modules (math, random, statistics)

Built-in functions:

Python provides many functions out-of-the-box. For example: print(), input(), abs(), divmod(), max(), min(), pow(), sum(), etc. Using the standard library helps you avoid "reinventing the wheel".

Input / Output:

- input() reads a line from the user (as string).
- print() outputs to console.

  Example:

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

Mathematical functions:

Here are some useful ones:

- abs(x) → absolute value of x.
  Example: abs(-5) is 5.
- divmod(a, b) → returns tuple (a // b, a % b).
  Example: divmod(17, 5) → (3, 2).
- max(iterable) / min(iterable) → largest/smallest in an iterable.
  Example: max([3, 7, 2]) → 7.
- pow(x, y) → x raised to power y.
  Example: pow(2, 3) → 8. (Note: there is also built-in ** operator).
- sum(iterable) → sum of numbers in iterable.
  Example: sum([1,2,3]) → 6.

Modules:

Modules are separate files/packages of code that you can import to gain extra functionality.

- import math → gives access to many mathematical features (e.g., math.sqrt(), math.pi, etc).
- import random → for randomness: random.randint(), random.choice(), etc.
- import statistics → for statistical functions: statistics.mean(), statistics.median(), etc.

Example usage:

```
import math, random

print(math.pi)
print(random.randint(1, 10))
```

Why use modules and standard library?

- They save time and effort.
- They are well-tested and reliable.
- They enable writing more powerful programs with less code.

Chapter 5: Strings and Lists

5.1 Introduction to Strings, String Operations, Traversing a String

Strings:

A string is a sequence of characters. In Python, anything within quotes ('...' or "..." or triple quotes) is a string.

Example:

```
s1 = "Hello"
s2 = 'World'
print(s1, s2)
```

String operations:

- **Indexing:** You can access characters by position (starting from 0).
  Example: s1[1] gives 'e'.
- **Slicing:** Extract a substring using [start:end] (end excluded).
  Example: s1[1:4] gives 'ell'.
- **Concatenation:** Use + to join strings.
  Example: "Hello" + " " + "World" → "Hello World".
- **Repetition:** Use * to repeat a string.
  Example: "ha" * 3 → "hahaha".
- **Traversing:** Loop through each character using a for loop.
  Example:

```
for ch in "hello":
    print(ch)
```

Immutability of strings:

Strings cannot be changed once created (they are immutable). You can build new strings, but you cannot modify a string in-place. Example: s[0] = 'H' is invalid.

5.2 String Methods and Built-in Functions

Python provides many built-in methods for string objects. Some commonly used ones:

- len(s) → length of string s. Example: len("abc") → 3.
- s.upper() / s.lower() → convert to uppercase/lowercase.
- s.strip() → remove whitespace from both ends.
- s.replace(old, new) → replace substring.
- s.split(separator) → split into list of substrings.
- s.find(sub) → find first occurrence of sub, give index or -1 if not found.
- s.isnumeric() / s.isalpha() etc → check type of characters.

Example:

```
text = "  Hello World  "
print(text.strip())            # "Hello World"
print(text.upper())            # "  HELLO WORLD  "
print("apple banana".split())  # ['apple', 'banana']
```

5.3 Introduction to List and its Operations

What is a list?

A list is an ordered, mutable collection of items. It can hold elements of different types (integers, strings, other lists, etc).

Example of creating lists:

```
a = [1, 2, 3, 4, 5]
b = ['apple', 'banana', 'cherry']
c = [1, 'hello', 3.14, True]
```

Accessing list elements:

- Indexing: a[0] is the first element.
- Negative indexing: a[-1] is the last element.
- Slicing: a[1:4] gives elements from index 1 up to index 3.

Adding and updating elements:

- append(item) adds at end.
- insert(index, item) inserts at a given position.
- extend(iterable) adds multiple items.
- Since lists are mutable, you can assign: a[1] = 25.

Removing elements:

- remove(item) removes first matching item.
- pop(index) removes item at index and returns it.
- del a[index] deletes element at index.

Iteration:

You can loop through lists using for:

```
for fruit in ['apple', 'banana', 'cherry']:
    print(fruit)
```

5.4 List Methods and Built-in Functions ● Nested and Copying Lists

List methods / built-in functions:

Some useful methods/functions:

- len(list) → number of elements.
- list.append(x), list.insert(i, x), list.extend([...]) as above.
- list.remove(x), list.pop(), list.clear() → empty list.

- list.sort() → sort in-place.
- sorted(list) → returns sorted list without changing original.
- list.reverse() → reverse in-place.
- list.copy() → returns a shallow copy of the list.

Nested lists:

A list can contain another list as an element. Hence you can have "list of lists" (useful for tables/matrices). Example:

```
matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
print(matrix[1][2])   # accesses the element at row-1, column-2 → 6
```

Copying lists – shallow vs deep copy:

- If you do new_list = original_list, both names refer to the *same* list object.
- If you do new_list = original_list.copy() (or use slicing [:]), you make a new list, but the elements inside (if they are mutable) are still references to the same objects (so it's a shallow copy).
- For nested lists you may need a *deep copy* (using copy.deepcopy() from the copy module) so that inner lists are duplicated not just referenced.

Example:

```
orig = [[1,2], [3,4]]
shallow = orig.copy()
shallow[0][0] = 99
print(orig)     # will show [[99,2], [3,4]] because inner lists were shared
```

5.5 List as Arguments to Function

Lists can be passed as arguments to functions, just like other objects. Because lists are mutable, a function can modify the list passed to it (unless you purposely avoid that). Example:

```
def add_item(my_list, item):
  my_list.append(item)

fruits = ['apple', 'banana']
add_item(fruits, 'cherry')
print(fruits)     # ['apple', 'banana', 'cherry']
```

Key points to remember:

- Since lists are mutable and are passed by reference (effectively), modifications inside the function reflect outside unless you create a fresh list inside the function.
- If you want to avoid modifying the original, you might make a copy inside the function: local_copy = my_list.copy() then work with local_copy.