TAPC2411 Professional Course on Python Programming

TechZ Academy

August 12, 2025

Contents

Chapter 1

Introduction to Python Programming

1.1 Introduction

1.1.1 Programming Language

A programming language is a formal language comprising a set of instructions that produce various kinds of output. It allows programmers to communicate instructions to a computer system, enabling it to perform specific tasks or functions. Programming languages are used to create software, websites, applications, and other computational solutions.

Examples of programming languages include `Python`, `Java`, `C++`, `JavaScript`, `Ruby`, and many more. Each language has its own syntax, semantics, and rules that govern how instructions are written and executed.

1.1.2 Algorithms

An **algorithm** is a step-by-step procedure or set of rules designed to solve a specific problem or perform a particular task. It is a finite sequence of well-defined, unambiguous instructions that, when followed, lead to a desired outcome or solution. Algorithms can be expressed in various forms, including natural language, pseudocode, flowcharts, or programming languages.

In computer science, algorithms are fundamental to solving computational problems efficiently. They provide a blueprint for writing computer programs and are essential for tasks such as sorting data, searching for information, performing mathematical operations, and optimizing processes.

Key characteristics of algorithms:

1. **Input:** The algorithm receives data or input values on which it operates.

2. **Output:** The algorithm produces a result or output based on the input and the operations performed.

3. **Determinism:** Each step of the algorithm is precisely defined and produces the same result when executed with the same input.

4. **Finiteness:** The algorithm has a finite number of steps, meaning it eventually terminates.

5. **Effectiveness:** Each step of the algorithm must be clear and executable within finite time and space constraints.

Algorithms are used extensively in various fields, including computer science, mathematics, engineering, and everyday problem-solving tasks. They serve as the foundation for designing efficient software systems, algorithms for data analysis, artificial intelligence, and more.

1.1.3 Types of Programming Language

Programming languages can be categorized into several types based on various criteria such as their level of abstraction, paradigm, purpose, and domain of application. Here are some common types:

1. **Low-level languages:**

   - **Machine Language:** The lowest-level programming language consisting of binary code understood directly by the computer's hardware.
   - **Assembly Language:** Uses mnemonic codes to represent machine instructions, making it more readable than machine language but still closely tied to the hardware architecture.

2. **High-level languages:**

   - **Procedural Languages:** Focus on describing a sequence of steps to solve a problem. Examples: `C`, `Pascal`, `BASIC`.
   - **Object-Oriented Languages:** Organize code into objects that interact with each other, emphasizing encapsulation, inheritance, and polymorphism. Examples: `Java`, `C++`, `Python`.
   - **Functional Languages:** Treat computation as the evaluation of mathematical functions and emphasize immutability and declarative programming. Examples: `Haskell`, `Lisp`, `Scala`.

- **Scripting Languages:** Designed for quick and easy development of small to medium-sized programs. Examples: `Python`, `Ruby`, `JavaScript`.

3. **Domain-specific languages (DSLs):**

   - **Markup Languages:** Used to annotate text for formatting or semantic information. Examples: `HTML`, `XML`, `Markdown`.
   - **Query Languages:** For querying and manipulating databases or data sources. Examples: `SQL`, `XPath`.
   - **Statistical Languages:** For statistical analysis and data manipulation. Examples: `R`, `MATLAB`.

4. **Functional purpose:**

   - **General-purpose languages:** Versatile and used for a wide range of applications. Examples: `Python`, `Java`, `C++`.
   - **Domain-specific languages:** Tailored to a specific application domain. Examples: `SQL`, `HTML/CSS`, `VHDL`.

5. **Compiled vs. interpreted languages:**

   - **Compiled Languages:** Code is translated into machine code before execution. Examples: `C`, `C++`, `Rust`.
   - **Interpreted Languages:** Code is executed line by line by an interpreter. Examples: `Python`, `Ruby`, `JavaScript`.

6. **Imperative vs. Declarative languages:**

   - **Imperative Languages:** Programs are composed of statements that change a program's state. Examples: `C`, `Java`, `Python`.
   - **Declarative Languages:** Programs describe the desired result without specifying the exact steps. Examples: `SQL`, `HTML/CSS`, `Prolog`.

1.2 Compilers

1. **Lexical Analysis (Scanning):**

   - Breaks the source code into tokens or lexemes.
   - The lexer (scanner) reads the source code character by character and groups them into tokens.
   - Comments and whitespace are usually discarded.

2. **Syntax Analysis (Parsing):**
   - Analyzes the structure of the source code to ensure it conforms to syntax rules.
   - The parser uses a grammar specification to check if the tokens form valid statements and expressions.
   - Generates a parse tree or abstract syntax tree (AST).
3. **Semantic Analysis:**
   - Verifies the meaning of the source code beyond its syntax.
   - Checks for semantic errors, such as type mismatches or undefined variables.
   - Involves symbol table management.
4. **Intermediate Code Generation:**
   - Translates the source code into an intermediate representation (IR).
   - May involve translating the AST into a lower-level representation.
5. **Optimization:**
   - Applies optimization techniques to improve code efficiency.
   - Examples: constant folding, dead code elimination, loop optimization.
6. **Code Generation:**
   - Translates the optimized intermediate code into machine code.
   - Register allocation and instruction scheduling may occur.
7. **Linking (for multi-file programs):**
   - Combines object files into a single executable or library.
   - Resolves external references and performs address binding.

Each stage of the compiler transforms the source code into executable code efficiently. The resulting executable can then be run on the target platform.

1.3 Interpreter Language

1. **Lexical Analysis:**
   - Reads the source code character by character and groups them into tokens.
   - Tokens represent the smallest units, such as keywords, identifiers, operators, and literals.

- Comments and whitespace are typically ignored.

2. **Parsing:**
   - Analyzes the structure of the source code to ensure it conforms to syntax rules.
   - Checks if the tokens form valid statements and expressions.
   - May generate an abstract syntax tree (AST).

3. **Semantic Analysis:**
   - Verifies the meaning of the source code beyond its syntax.
   - Checks for semantic errors.
   - Involves symbol table management.

4. **Code Execution:**
   - Executes the parsed code directly without a separate code generation stage.
   - Interprets the AST or intermediate representation.
   - May use strategies like bytecode interpretation or just-in-time (JIT) compilation.

Unlike compilers, interpreters do not generate machine code but execute the code directly.

Table 1.1: Difference Between Compiler and Interpreter

Compiler	**Interpreter**
Translates entire program to machine code before execution.	Translates code line by line and executes it simultaneously.
Produces intermediate object code or executable file.	Does not produce an intermediate object code or executable file.
Typically generates faster-executing code.	Generally slower as it executes code directly.
Detects all syntax and semantic errors before execution.	May detect errors at runtime during interpretation.
Requires a separate compilation stage before execution.	Does not require a separate compilation stage.
Examples include GCC (GNU Compiler Collection), Clang.	Examples include Python interpreter, JavaScript interpreter.

Table 1.2: Differences Between Procedural, Object-Oriented, and Functional Programming

Procedural Programming	**Object-Oriented Programming**	**Functional Programming**
Focuses on procedures or functions that operate on data.	Organizes code into objects that encapsulate data and behavior.	Emphasizes mathematical functions and immutable data.
Data and procedures are separate entities.	Data and behavior are encapsulated within objects.	Functions are first-class citizens and can be passed as arguments or returned.
Languages: C, Pascal.	Languages: Java, C++, Python.	Languages: Haskell, Lisp, Clojure.
Follows a top-down approach to problem-solving.	Follows a bottom-up approach with focus on reusability and modularity.	Follows a declarative approach; functions describe what should be done.
Less emphasis on code reusability and scalability.	Promotes code reusability through inheritance, polymorphism, encapsulation.	Encourages code reusability through higher-order functions and immutability.

Chapter 2

Conditional Statements in Python

Conditional statements let your Python programs make decisions and control the flow of execution based on whether certain conditions are **true** or **false**. They are a fundamental part of programming and real-world problem solving.

2.1 Introduction to Conditional Statements

Python provides several types of conditional statements:

- `if`
- `if-else`
- `if-elif-else`
- Nested `if-else`

Note: Indentation in Python

Indentation in Python is critical. It shows which lines of code belong to which code blocks.

- **Default indentation**: 1 tab *or* 4 spaces (may vary by system).
- **Rule**: Use consistent indentation in the same block.

Example:

```
if condition:
    statement1
    statement2
```

Here, `statement1` and `statement2` will be executed only if the condition is true.

2.2 The `if` Statement

The simplest decision-making statement in Python is the `if` statement.

Syntax

```
if <relational expression>:
    # Block of code (executed if condition is True)
    Line 1
    Line 2
    ...
```

Example 1: Checking Zero

```
number = input("Enter the number: ")
# The input function returns a string.
number = int(number)     # Convert to integer

if number == 0:
    print("Entered Number is ZERO")
```

If the user enters 0, "Entered Number is ZERO" is printed.

2.3 Truthy and Falsy Values

In Python, certain values are considered "true" or "false" in conditions (e.g., 0 → False, nonzero → True).

Example 2: Truthy/Falsy Check

```
number = input("Enter the number: ")
number = int(number)

if number:
    print("Entered Number is not ZERO")

if not number:
    print("Entered Number is ZERO")
```

When `number` is 0, only the second block executes. Otherwise, the first block executes.

2.4 The `if-else` Statement

The `if-else` statement allows one block (if) when true, another (`else`) when false.

Syntax

```
if <condition>:
    # true block
else:
    # false block
```

Example 3: Zero or Not

```
number = input("Enter the number: ")
number = int(number)
if number == 0:
    print("Entered Number is ZERO")
else:
    print("Entered Number is not ZERO")
```

2.5 The `if-elif-else` Statement

For multiple conditions, use `if-elif-else`.

Syntax

```
if <condition1>:
    # block 1
elif <condition2>:
    # block 2
else:
    # block 3
```

Example 4: Zero, One, or Others

```
number = input("Enter the number: ")
number = int(number)

if number == 0:
    print("Entered Number is ZERO")
elif number == 1:
    print("Entered Number is ONE")
else:
    print("Others")
```

2.6 Practical Examples

Example 5: Multiple Divisibility Checks

```
number = input("Enter the number: ")
number = int(number)

if number % 2 == 0:
    print("Twilight")
elif number % 3 == 0:
    print("SQL")
else:
    print("PYTHON")
```

Checks for divisibility by 2 (Twilight), 3 (SQL), or otherwise outputs PYTHON.

2.7 Nested `if-else` Statements

Statements can be nested for more complex logic.

Syntax

```
if <condition1>:
    if <condition2>:
        # Block A
    else:
        # Block B
else:
    # Block C
```

Example 6: Multiple Divisibility with Nesting

```
number = input("Enter the number: ")
number = int(number)

if number % 2 == 0:
    if number % 3 == 0:
        print("The number is divisible by 6")
    elif number % 5 == 0:
        print("The number is divisible by 10")
```

Checks for divisibility by 6 or 10, only if the number is even.

2.8 Summary

- `if`: For single conditions.
- `if-else`: For two-way branching.
- `if-elif-else`: For multiple branches.
- **Indentation** is essential in Python for defining code blocks.
- Use nesting for complex decision logic.

Practice writing conditional statements to master this essential concept!

Exercises

1. Write a program that accepts marks from the user and prints “Pass” if marks are greater than or equal to 35, otherwise prints “Fail”.
2. Write a program to check if a number is positive, negative, or zero.
3. Write a program to check if a number is odd, even, or zero.

Conditional statements give Python programs the ability to make decisions and adapt to different situations. Master them to become a better Python programmer!

Chapter 3

List

3.1 Introduction to Lists

A **list** in Python is a collection of data in an ordered fashion.

- **Collection of data**: Lists can hold heterogeneous data; different elements of various data types can be grouped into a single list.
- **Ordered Fashion**: Lists are indexed; the input order and storing order will be the same and remain unaltered by the interpreter.

3.1.1 Indexing in Lists

Python follows **zero-based indexing**:

- Positive indexing: From 0 to $n - 1$, where n is the length of the list.
- Negative indexing: From $-n$ to -1, where -1 refers to the last element.

3.2 Creating a List

Lists are created using square brackets `[ ]`.
Syntax:

```
list_variable = [value1, value2, ...]
```

Example:

```
l = [1, 2, 3, 4, "a", "e", "i", "o", "u", True]
vowels = ["a", "e", "i", "o", "u"]
```

Storage Representation:

Index	0	1	2	3	4
Element	"a"	"e"	"i"	"o"	"u"
Negative Index	-5	-4	-3	-2	-1

3.2.1 Index Conversion

- Negative to Positive: Add n to the index.
- Positive to Negative: Subtract n from the index.

Example:

```
vowels[-1] == vowels[-1 + 5]
vowels[1] == vowels[1 - 5]
```

3.3 Accessing List Elements

Elements can be accessed using subscripting:

```
vowels[0]  # First element
vowels[-1] # Last element
```

3.4 Slicing of Lists

Slicing extracts a portion of a list:

```
list_variable[m:n]        # Elements from index m to n-1
list_variable[m:n:step]   # With step size
```

Examples:

```
vowels = ["a", "e", "i", "o", "u"]
vowels[1:4]      # ['e', 'i', 'o']
vowels[-4:-1]    # ['e', 'i', 'o']
vowels[3:]       # ['o', 'u']
vowels[:2]       # ['a', 'e']
vowels[::2]      # ['a', 'i', 'u']
vowels[::-1]     # ['u', 'o', 'i', 'e', 'a']
```

Note: Step size cannot be zero.

3.5 Mutability of Lists

Lists are mutable, meaning their elements can be altered (CRUD operations):

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

Example:

```
vowels[0] = "p"
```

3.6 List Functions and Methods

3.6.1 len()

Returns the number of elements in a list.

```
len(vowels)
```

3.6.2 dir()

Returns all attributes and methods of the object.

```
dir(vowels)
```

3.6.3 sort()

Sorts the list in ascending or descending order.

```
vowels.sort()
vowels.sort(reverse=True)
```

3.6.4 pop()

Removes and returns the last element (or specified index).

```
vowels.pop()
```

3.6.5 append()

Adds an element to the end of the list.

```
vowels.append("z")
```

3.6.6 insert()

Inserts an element at a given index.

```
vowels.insert(1, "b")
```

3.6.7 remove()

Removes the first occurrence of the specified value.

```
vowels.remove("a")
```

3.7 for Loop with Lists

Example: Creating a list of even numbers till n.

```
n = int(input("Enter n: "))
list_even = []
for i in range(0, n+1, 2):
list_even.append(i)
print(list_even)
```

Chapter 4

Tuples

4.1 Introduction

A **tuple** in Python is an ordered, immutable collection of elements. Unlike lists, tuples cannot be modified once created. This immutability makes tuples useful for storing fixed sets of values and for use as keys in dictionaries (if they contain only immutable elements).

Tuples are defined by enclosing elements in `()` parentheses, separated by commas.

4.2 Creating Tuples

You can create tuples in several ways:

```
# Empty tuple
t1 = ()

# Tuple with multiple elements
t2 = (1, 2, 3)

# Tuple without parentheses (tuple packing)
t3 = 1, 2, 3

# Single-element tuple (comma is required)
t4 = (1,)

# Using tuple() constructor
t5 = tuple([1, 2, 3])
```

4.3 Accessing Tuple Elements

Tuple elements can be accessed using indexing and slicing.

```
t = (10, 20, 30, 40, 50)

# Access first element
print(t[0])  # Output: 10

# Access last element
print(t[-1]) # Output: 50

# Slicing
print(t[1:4])  # Output: (20, 30, 40)
```

4.4 Tuple Immutability

Once created, tuple elements cannot be changed, added, or removed.

```
t = (1, 2, 3)
t[0] = 10  # Error: TypeError: 'tuple' object does not support item assignment
```

However, if a tuple contains mutable objects (like lists), those mutable objects can be modified.

```
t = ([1, 2], [3, 4])
t[0].append(5)
print(t)  # Output: ([1, 2, 5], [3, 4])
```

4.5 Tuple Operations

Tuples support many of the same operations as lists, except for those that modify the data.

4.5.1 Concatenation

```
t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)  # Output: (1, 2, 3, 4)
```

4.5.2 Repetition

```
t = (1, 2)
print(t * 3)  # Output: (1, 2, 1, 2, 1, 2)
```

4.5.3 Membership

```
t = (1, 2, 3)
print(2 in t)   # Output: True
print(5 not in t) # Output: True
```

4.6 Tuple Functions and Methods

- `len(tuple)`: Returns the number of elements in the tuple.
- `max(tuple)`: Returns the maximum element.
- `min(tuple)`: Returns the minimum element.
- `sum(tuple)`: Returns the sum of elements (numeric tuples only).
- `tuple.count(value)`: Counts the number of occurrences of `value`.
- `tuple.index(value)`: Returns the index of the first occurrence of `value`.

```
t = (10, 20, 30, 20)
print(len(t))      # Output: 4
print(max(t))      # Output: 30
print(min(t))      # Output: 10
print(t.count(20)) # Output: 2
print(t.index(30)) # Output: 2
```

4.7 Tuple Packing and Unpacking

Tuples allow **packing** multiple values into one variable and **unpacking** them back into separate variables.

```
# Packing
t = 1, 2, 3

# Unpacking
a, b, c = t
print(a, b, c)  # Output: 1 2 3
```

4.8 Nested Tuples

Tuples can contain other tuples as elements.

```
t = (1, (2, 3), (4, 5, 6))
print(t[1])    # Output: (2, 3)
print(t[1][0]) # Output: 2
```

4.9 Use Cases of Tuples

1. Storing related data that should not change.
2. As keys in dictionaries (if all elements are immutable).
3. Returning multiple values from a function.
4. Fixed-size records in databases.

4.10 Summary

- Tuples are ordered, immutable collections.
- Defined using parentheses or without parentheses (tuple packing).
- Elements are accessed via indexing and slicing.
- Support concatenation, repetition, and membership checks.
- Often used for fixed, unchangeable collections of data.

Chapter 5

Sets and Dictionaries

5.1 Introduction

A **dictionary** in Python is a collection of **key-value pairs**. Each **key** is unique and used to access its corresponding **value**. Dictionaries are **mutable**, meaning they can be updated, added to, or cleared after creation.

5.2 Creating a Dictionary

Dictionaries can be created using curly braces {} or the `dict()` constructor.

```
# Example dictionary
d = {"a": 1, "e": 2, "i": 3, "o": 4, "u": 5}
```

Interactive Output:

```
>>> d
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

5.3 Accessing Dictionary Keys

```
>>> d.keys()
dict_keys(['a', 'e', 'i', 'o', 'u'])
>>> type(d.keys())
<class 'dict_keys'>
```

Converting to a list:

```
>>> ls = list(d.keys())
>>> ls
['a', 'e', 'i', 'o', 'u']
```

5.4 Adding New Key-Value Pairs

```
>>> d["z"] = 9
>>> d
{’a’: 1, ’e’: 2, ’i’: 3, ’o’: 4, ’u’: 5, ’z’: 9}
```

5.5 Live Views of Keys and Values

When you store the result of `.keys()` or `.values()`, the object is a **live view** of the dictionary. Changes to the dictionary are automatically reflected.

```
>>> l = d.keys()
>>> d["y"] = 10
>>> l
dict_keys([’a’, ’e’, ’i’, ’o’, ’u’, ’z’, ’y’])
```

Tip:

- `list(d.keys())` → static copy
- `d.keys()` → live view

5.6 Dictionary Methods

Method	Description	Example
`keys()`	Returns a view of all keys	`d.keys()`
`values()`	Returns a view of all values	`d.values()`
`items()`	Returns (key, value) pairs	`d.items()`
`get(key, default)`	Safely get a value	`d.get(’a’, 0)`
`fromkeys(iterable, value)`	Create dict from keys with same value	`dict.fromkeys([’a’,’b’]`
`update(other_dict)`	Add/overwrite from another dict	`d.update({’x’: 6})`
`pop(key)`	Remove and return value for key	`d.pop(’a’)`
`popitem()`	Remove and return last inserted pair	`d.popitem()`
`clear()`	Remove all items	`d.clear()`

5.7 Looping Through Dictionaries

```
# Keys only
for key in d:
print(key)
```

```
# Keys and values
for key, value in d.items():
print(key, value)
```

5.8 Advanced Dictionary Operations

5.8.1 Dictionary Comprehensions

```
# Squares of numbers from 1 to 5
squares = {x: x**2 for x in range(1, 6)}
print(squares)
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

5.8.2 Conditional Dictionary Comprehensions

```
# Only even squares
even_squares = {x: x**2 for x in range(1, 6) if x % 2 ==
   0}
# Output: {2: 4, 4: 16}
```

5.8.3 Merging Dictionaries

Python 3.9+:

```
d1 = {"a": 1, "b": 2}
d2 = {"b": 3, "c": 4}
merged = d1 | d2
# Output: {'a': 1, 'b': 3, 'c': 4}
```

Before Python 3.9:

```
merged = {**d1, **d2}
```

5.8.4 Nested Dictionaries

```
students = {
        "Alice": {"math": 90, "science": 85},
        "Bob": {"math": 78, "science": 82}
}
print(students["Alice"]["math"])  # Output: 90
```

5.8.5 Inverting Dictionaries

```
d = {"a": 1, "b": 2, "c": 3}
inv = {v: k for k, v in d.items()}
print(inv)
# Output: {1: 'a', 2: 'b', 3: 'c'}
```

5.8.6 Using `defaultdict` for Automatic Values

```
from collections import defaultdict
dd = defaultdict(int)
dd["x"] += 1
print(dd)
# Output: defaultdict(<class 'int'>, {'x': 1})
```

5.8.7 Sorting Dictionaries

```
# Sort by keys
sorted_by_keys = dict(sorted(d.items()))

# Sort by values
sorted_by_values = dict(sorted(d.items(), key=lambda item
    : item[1]))
```

5.9 Summary

- Dictionaries store unique keys with associated values.
- `.keys()`, `.values()`, and `.items()` return live views.
- Keys must be immutable, values can be any type.
- Advanced features include comprehensions, merging, nested structures, and sorting.

Chapter 6

Functions in Python

6.1 Introduction to Functions

In Python, a **function** is a block of reusable code designed to perform a specific task. Once defined, a function can be invoked multiple times within a program, avoiding code repetition and improving readability.

Functions follow the **Input** → **Process** → **Output** paradigm:

- **Input**: Data provided to the function (parameters/arguments)
- **Process**: Logic implemented inside the function
- **Output**: Result returned by the function

Key perspectives of functions:

1. **Function Definition** – How a function is created.
   - Input: **Parameters** (also called formal parameters)
   - Output: **Return value**
2. **Function Call** – How a function is invoked.
   - Input: **Arguments** (also called actual parameters)
   - Output: The return value is stored in a variable or used directly.

6.2 Defining a Function

The general syntax of defining a function in Python is:

```
def function_name(parameters):
# Code block
return value
```

6.3 Example 1 – Adding Two Numbers

```
def add_numbers(p, q):
sum = p + q
return sum
```

Function Call Statements:

Method 1: Using variables

```
a = 5
b = 10
result = add_numbers(a, b)
print(f"The sum of {a} and {b} is: {result}")
```

Method 2: Directly in the print statement

```
print("The sum of %d and %d is: %d" % (a, b, add_numbers(a, b)))
```

6.4 Parameters and Arguments

- **Parameters**: Variables declared in the function definition (placeholders).
- **Arguments**: Actual values passed to the function during a call.

Example: In `add_numbers(p, q)`, `p` and `q` are parameters. In `add_numbers(a, b)`, `a` and `b` are arguments.

Rules for passing arguments:

1. The number of arguments must match the number of parameters.
2. The order of arguments must match the order of parameters.
3. Arguments can be passed by position or by keyword.

6.5 Example 2 – Subtracting Two Numbers

```
def subtract_numbers(p, q):
difference = p - q
return difference
```

Function Call Examples:

```
x = 15
y = 5
result_subtract = subtract_numbers(x, y)
print(f"The difference between {x} and {y} is: {result_subtract}")

>>> subtract_numbers(15, 5)
10
>>> subtract_numbers(5, 15)
-10
>>> subtract_numbers(q=5, p=15)  # Keyword arguments
10
```

6.6 Default Parameters

A parameter can be assigned a default value, making it optional during a function call.

```
def subtract_numbers(p, q=1):
difference = p - q
return difference
```

6.7 Practical Example – Even/Odd Check with Multiple Functions

```
def two_input():
"""Receive two inputs from the user, convert to integers, and return."""
a = input("Enter a: ")
b = input("Enter b: ")
return int(a), int(b)

def even_check(n):
if n % 2 == 0:
return True
else:
return False

def core_function(a, b):
if even_check(a) and even_check(b):
return a * b
else:
```

```
return a / b

if __name__ == "__main__":
p, q = two_input()
print(core_function(p, q))
```

6.8 Recursion – Factorial Example

A function can call itself (**recursion**). Example: Calculating factorial of a number.

```
def facto(n):
if n == 1:
return 1
else:
return n * facto(n - 1)

if __name__ == "__main__":
i = int(input("Enter n: "))
print(facto(i))
```

6.9 Optimizing Recursion – Memoization Example

Memoization stores already computed results to improve performance.

```
fact = {0: 1, 1: 1}

def iskey(d, element):
list_of_keys = list(d.keys())
return element in list_of_keys

def facto(n):
if iskey(fact, n):
return fact[n]
else:
t = n * facto(n - 1)
fact[n] = t
return t

if __name__ == "__main__":
i = int(input("Enter n: "))
```

```
print(facto(i))
```

6.10 Summary

- Functions improve **code reuse** and **readability**.
- Functions have **parameters** (placeholders) and **arguments** (actual values).
- Arguments can be passed by **position** or **keyword**.
- Functions can be recursive and can use **memoization** for efficiency.
- Default parameters make function calls more flexible.

6.11 Assignment

1. Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure `BUBBLESORT` sorts array $A[1 : n]$.

```
BUBBLESORT(A, n)
1   for i = 1 to n - 1
2       for j = n downto i + 1
3           if A[j] < A[j - 1]
4               exchange A[j] with A[j - 1]
```

2. Suppose we want to sort the list: [8, 4, 6, 2] in ascending order.

   - Start with the first element (8). A single element is always considered sorted. Sorted portion: [8]
   - Take the next element (4). Compare it with 8. Since 4 is smaller, insert it before 8. Sorted portion: [4, 8]
   - Take the next element (6). Compare it with 8 ($6 < 8$, so move 8 one step right). Compare it with 4 ($6 > 4$, so stop). Insert 6 between 4 and 8. Sorted portion: [4, 6, 8]
   - Take the next element (2). Compare it with 8, then 6, then 4 (2 is smaller than all of them). Move each one step to the right and insert 2 at the start. Sorted portion: [2, 4, 6, 8]

   Implement a Python Code for implementing the above insertion sort algorithm.

3. Convert the above algorithm to sort the items in descending order.

4. Convert the program from Question 2, to handle both the ascending and descending order based on a parameter reverse whose default value is False.

5.