WELCOME TO

The Alphabet of Python

– AUTHOR
K.V.K.Phani Kumar

About the Author

Kodukulla Venkata Kameswara Phani Kumar

Degrees :-

- Diploma in Computer Science and Engineering
- B.Tech in Computer Science and Engineering
- Hindu Priest

—----------

I am a passionate educator and Python developer who combines technical expertise with a deep commitment to teaching excellence. Currently pursuing my Bachelor of Technology in Computer Science and Engineering, I have already distinguished myself as a dedicated contributor to programming education.

With a solid foundation built through my Diploma in Computer Science and Engineering and extensive hands-on experience in Python programming, I bring both academic knowledge and practical insights to my educational approach. My unique background as a Hindu priest adds a distinctive perspective to my teaching philosophy, emphasizing patience, clarity, and the noble pursuit of knowledge sharing.

Mission and Vision

My inspiration for writing "The Alphabet of Python" stems from my genuine passion for teaching and my unwavering commitment to providing high-quality education to aspiring programmers. I firmly believe that access to well-structured, valuable educational content can transform lives and open doors to countless opportunities in the technology sector.

My primary goal is to establish a meaningful standard of educational excellence that creates lasting value for the programming community. Through my work, I seek to build recognition while contributing positively to society by empowering individuals with essential programming skills that can enhance their careers and personal growth.

Teaching Philosophy

My educational approach is rooted in the belief that complex programming concepts can be made accessible to learners of all backgrounds when presented with clarity and patience. My passion for writing and teaching drives me to create resources that not only educate but also inspire confidence in readers as they embark on their programming journey.
Drawing from my spiritual background, I emphasize the importance of dedication, practice, and mindful learning. I understand that mastering programming requires both theoretical understanding and practical application, and this philosophy is reflected throughout "The Alphabet of Python."

About This Work

"The Alphabet of Python" represents my dedication to delivering exceptional educational content that bridges the gap between beginner curiosity and professional competence. The book reflects my understanding that effective programming education must balance comprehensive coverage with practical applicability, all while maintaining an approachable and encouraging tone for learners at every level.

About "The Alphabet in Python"

"The Alphabet in Python" is your complete guide to mastering Python programming from absolute beginner to building advanced AI applications. This comprehensive book combines solid theoretical foundations with hands-on practical experience, making it the perfect resource for anyone looking to become proficient in Python.

What's Inside

The book systematically covers all essential Python concepts in a logical progression: Introduction to Python, Environment Setup, Variables and Data Types, Operators, Conditional Statements, Loops, Functions, Lists, Tuples, Sets, Dictionaries, Strings, Input/Output, Exception Handling, Modules and Packages, File Handling, Object-Oriented Programming, and Built-in Functions.
But this isn't just another theory book. You'll find over 25 practical projects that let you apply what you learn immediately, building real applications that demonstrate Python's versatility in solving everyday problems.

Special Feature: Complete Jarvis AI Assistant

As a bonus, the book includes the complete source code for building your own Jarvis AI Assistant integrated with Google's Gemini API. This advanced project shows you how to create a voice-activated AI assistant that can understand natural language, respond intelligently, and perform various tasks—bringing together everything you've learned in an impressive, real-world application.

Perfect For Everyone

Whether you're a complete programming beginner, a student, a career switcher entering tech, or a professional familiar with other languages, this book meets you where you are. The clear explanations and step-by-step approach make complex concepts accessible, while the progressive project structure builds your confidence through practical application.

What You'll Master

By completing this book, you'll confidently write clean Python code, build complex applications using object-oriented programming, handle data with Python's powerful built-in structures, integrate AI services into your applications, and develop the problem-solving mindset essential for real-world programming challenges.

Learning Approach

Every concept is reinforced through clear explanations, working code examples, immediate practice exercises, and progressively challenging projects. This hands-on methodology ensures you don't just learn Python syntax—you develop the practical skills and confidence to tackle any Python project.
Ready to transform from Python novice to confident developer? Your comprehensive journey starts here.

References

Official Python Documentation and Resources

Core Documentation

- Python Official Documentation: https://docs.python.org/3/
- Python Tutorial: https://docs.python.org/3/tutorial/
- Python Language Reference: https://docs.python.org/3/reference/
- Python Standard Library: https://docs.python.org/3/library/
- Python Setup and Usage: https://docs.python.org/3/using/

Installation and Tools

- Python Downloads: https://www.python.org/downloads/
- pip Documentation: https://pip.pypa.io/en/stable/
- PyPI - Python Package Index: https://pypi.org/
- Virtual Environments (venv): https://docs.python.org/3/library/venv.html
- IDLE Documentation: https://docs.python.org/3/library/idle.html

Popular Libraries

- NumPy: https://numpy.org/doc/
- Pandas: https://pandas.pydata.org/docs/
- Matplotlib: https://matplotlib.org/stable/
- Requests: https://requests.readthedocs.io/
- Flask: https://flask.palletsprojects.com/
- Django: https://docs.djangoproject.com/

Community and Learning

- Python.org Community: https://www.python.org/community/
- Python Beginner's Guide: https://wiki.python.org/moin/BeginnersGuide
- Python FAQ: https://docs.python.org/3/faq/
- Python Success Stories: https://www.python.org/success-stories/
- Python Developer's Guide: https://devguide.python.org/

Note: All links are official Python Software Foundation resources. Always refer to official documentation for the most current information.

Motivation for Students

Inspirational Quotes from Tech Leaders

Stephen Hawking

Position: Theoretical Physicist, Cosmologist, Author
Education: BA in Natural Sciences (University of Oxford), PhD in Applied Mathematics and Theoretical Physics (University of Cambridge)
"Whether you want to uncover the secrets of the universe, or you just want to pursue a career in the 21st century, basic computer programming is an essential skill to learn."

Sundar Pichai

Position: CEO of Alphabet Inc. and Google
Education: B.Tech in Metallurgical Engineering (IIT Kharagpur), MS in Materials Science and Engineering (Stanford University), MBA (Wharton School, University of Pennsylvania)
"Moving beyond code and intensive degrees to these constant, lightweight and ubiquitous forms of education will take resources and experimentation."
"The core of Google's success is its innovation."
"As a leader, it is important to not just see your success but focus on the success of others."
"Technology should do the hard work so people can do the things that make them the happiest in life."

Bill Gates

Position: Co-founder of Microsoft, Philanthropist, Author
Education: Attended Harvard University (dropped out to start Microsoft)
"Technology is just a tool. In terms of getting the kids working together and motivating them, the teacher is the most important."
"Your most unhappy customers are your greatest source of learning."
"We always overestimate the change that will occur in the next two years and underestimate the change that will occur in the next ten years."
"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

Key Takeaways for Programmers

From Stephen Hawking: Programming is not just a technical skill—it's a gateway to understanding the universe and succeeding in the modern world. Technology gives us unlimited possibilities when we use it to communicate and connect.
From Sundar Pichai: Innovation is the foundation of success in technology. True leadership means focusing on others' success, and continuous learning should be lightweight and accessible to everyone.
From Bill Gates: Technology is merely a tool—what matters is how we use it to empower people and solve problems. Learning from failures and thinking long-term are essential for technological progress.

Table of Contents

The Alphabet of Python

CHAPTER-1

Python Programming Language: A Comprehensive Introduction

What is Python?

Imagine you want to talk to your computer and tell it exactly what to do. Python is like a special language that both you and your computer can understand easily. It's a programming language that lets you write instructions for computers in a way that's almost like writing in plain English!
Think of Python as a translator between your ideas and what the computer needs to do. Whether you want to build a website, analyze data, create games, or even teach computers to think (artificial intelligence), Python can help you do it all.

Why is Python So Special?

1. Easy to Read and Write

Python code looks clean and simple. Here's what makes it special:

- No confusing symbols everywhere
- Uses simple English words
- Organizes code in a way that makes sense
- If you can read English, you can probably guess what Python code does!

2. Perfect for Beginners

- You can start writing useful programs on your first day
- Fewer rules to memorize compared to other languages
- Friendly error messages that actually help you fix problems
- Lots of free learning resources available online

3. Powerful for Professionals

- Used by companies like Google, Netflix, Instagram, and NASA
- Can handle simple scripts and complex applications equally well
- Huge collection of ready-made tools (libraries) available

- Strong community support worldwide

The Story Behind Python

The Creator

Python was created by Guido van Rossum, a Dutch programmer, during his Christmas holidays in 1989. He was working in Amsterdam and wanted to create a programming language that was:

- Easy to use
- Powerful enough for real work
- Fun to program with

Why is it Called "Python"?

Here's a fun fact: Python is NOT named after the snake! Guido named it after "Monty Python's Flying Circus," a British comedy show he enjoyed. He wanted a name that was short, unique, and a bit playful.

Python's Growth Over Time

- 1991: First public version released
- 2000: Python 2.0 - Added many new features
- 2008: Python 3.0 - Major update (this is what we use today)
- 2020: Python 2 officially retired
- Today: One of the world's most popular programming languages

What Can You Do with Python?

1. Web Development

- Build websites and web applications
- Create online stores, blogs, and social media platforms
- Popular tools: Django, Flask

2. Data Analysis

- Analyze business data to make better decisions
- Create charts and graphs
- Process large amounts of information quickly
- Tools: Pandas, NumPy, Matplotlib

3. Artificial Intelligence & Machine Learning

- Teach computers to recognize images
- Build chatbots and recommendation systems
- Create predictive models
- Tools: TensorFlow, scikit-learn

4. Automation

- Automate boring, repetitive tasks
- Organize files and folders automatically
- Send emails or messages automatically
- Extract information from websites

5. Game Development

- Create simple games and interactive applications
- Build educational games
- Prototype game ideas quickly

6. Scientific Research

- Analyze scientific data
- Create simulations and models
- Process research results
- Used in fields like biology, physics, and astronomy

Python's Philosophy: Keep it Simple

Python follows a simple philosophy called "The Zen of Python". Here are the key ideas:

- Simple is better than complicated
- If it's hard to explain, it's probably not a good idea
- There should be one obvious way to do something
- Code should be easy to read

This means Python prioritizes making things clear and understandable rather than showing off with complex tricks.

The Python Community

One of Python's greatest strengths is its amazing community:

Helpful and Welcoming

- Millions of programmers worldwide use Python
- Always willing to help beginners

- Active forums and discussion groups

Tons of Resources

- Free tutorials and courses everywhere
- Thousands of ready-made code libraries
- Regular conferences and meetups (like PyCon)

Open Source

- Python is completely free to use
- Anyone can contribute to making it better
- Transparent development process

Who Uses Python?

Students: Perfect first programming language to learn
Web Developers: Build modern websites and applications
Data Scientists: Analyze data and create insights for businesses
Researchers: Process and analyze scientific data
Automation Engineers: Make repetitive tasks automatic
AI/ML Engineers: Build intelligent systems
System Administrators: Manage and automate computer systems

Getting Started: What You Need

The best part about Python? You need very little to get started:

1. A Computer: Windows, Mac, or Linux - Python works on all
2. Python Software: Free download from python.org
3. A Text Editor: Even simple ones work (though better tools make life easier)
4. Curiosity and Practice: The most important ingredients!

What Makes This Book Special

"The Alphabet of Python" is designed to take you from complete beginner to confident Python programmer:

- Step-by-step approach: Each chapter builds on the previous one
- Real examples: Every concept explained with practical code
- Exercises: Practice problems to reinforce learning
- Tips and tricks: Insider knowledge to make you more efficient
- Common mistakes: Learn from others' errors before making them yourself

Your Python Journey Starts Now

Whether you're a student learning your first programming language, a professional looking to add new skills, or someone curious about technology, Python welcomes you with open arms.

Remember: Every expert was once a beginner. The most important step is the first one, and you're about to take it!

In the next chapter, we'll get Python installed on your computer and write your very first program. Get ready to say "Hello, World!" in Python!

"Life is short, use Python!" - A popular saying in the Python community that captures the language's focus on productivity and enjoyment.

CHAPTER-2

Setting up Python Environment - Complete Guide

Python Installation

Windows Installation Download Python from the official website (python.org) and run the installer. During installation, ensure you check "Add Python to PATH" to make Python accessible from the command line. The installer includes pip (Python package installer) by default. Verify installation by opening Command Prompt and typing python --version and pip --version.
macOS Installation While macOS comes with Python pre-installed, it's recommended to install the latest version. Use the official installer from python.org or install via Homebrew with brew install python. Homebrew installation automatically handles PATH configuration and includes pip.
Linux Installation Most Linux distributions include Python by default. For Ubuntu/Debian systems, update to the latest version using sudo apt update && sudo apt install python3 python3-pip. For Red Hat/CentOS systems, use sudo yum install python3 python3-pip or sudo dnf install python3 python3-pip.

Development Environment Setup

Code Editors and IDEs Choose an appropriate development environment based on your needs:

- Visual Studio Code: Free, lightweight editor with excellent Python extension support, debugging capabilities, and integrated terminal
- PyCharm: Professional IDE with comprehensive Python support, intelligent code completion, and advanced debugging tools
- Jupyter Notebook: Interactive environment ideal for data science, research, and educational purposes
- Sublime Text/Atom: Lightweight editors with Python syntax highlighting and plugin support

Essential VS Code Extensions for Python Install the Python extension by Microsoft, which provides IntelliSense, debugging, code formatting, and linting. Additional useful extensions include Python Docstring Generator, Pylance for enhanced language support, and GitLens for version control integration.

Virtual Environments

Why Use Virtual Environments Virtual environments isolate project dependencies, preventing conflicts between different projects that may require different package versions. This ensures reproducible environments and cleaner project management.
Creating Virtual Environments Python includes the venv module for creating virtual environments. Navigate to your project directory and run python -m venv myenv to create a new environment. Activate it using myenv\Scripts\activate on Windows or source myenv/bin/activate on macOS/Linux. When activated, your command prompt will show the environment name.
Managing Dependencies With your virtual environment activated, install packages using pip: pip install package_name. Generate a requirements file with pip freeze > requirements.txt to document all installed packages. Other developers can recreate the environment using pip install -r requirements.txt.

Package Management with pip

Basic pip Commands

- pip install package_name: Install a package
- pip install package_name==version: Install specific version
- pip upgrade package_name: Update to latest version
- pip uninstall package_name: Remove a package
- pip list: Show installed packages
- pip show package_name: Display package information

Advanced Package Management Install packages from GitHub repositories using pip install git+https://github.com/user/repo.git. For development installations that reflect code changes immediately, use pip install -e . in the project directory. Create editable installations for packages you're actively developing.

Environment Variables and Configuration

Setting Environment Variables Configure Python-specific environment variables for better development experience. Set PYTHONPATH to include additional directories in the Python module search path. Use PYTHONDONTWRITEBYTECODE=1 to prevent Python from creating .pyc files during development.
Configuration Files Create .env files for project-specific environment variables and use the python-dotenv package to load them automatically. This approach keeps sensitive information like API keys separate from your code.

Development Tools and Best Practices

Code Formatting and Linting Install and configure development tools for code quality:

- Black: Automatic code formatter that enforces consistent style
- Flake8: Linting tool that checks for style and programming errors
- isort: Automatically sorts and organizes import statements
- mypy: Static type checker for Python code

Configure these tools in your IDE or set up pre-commit hooks to run them automatically before code commits.
Testing Environment Set up testing frameworks like pytest for writing and running tests. Create separate test environments and use tools like tox to test your code against multiple Python versions.
Version Control Integration Initialize Git repositories in your projects and create appropriate .gitignore files to exclude virtual environments, compiled Python files, and IDE-specific directories. Use meaningful commit messages and maintain clean project history.

Troubleshooting Common Issues

PATH Problems If Python commands aren't recognized, ensure Python is added to your system PATH. On Windows, use the "Add Python to PATH" option during installation or manually add the Python installation directory to your system environment variables.
Permission Issues On macOS/Linux, avoid using sudo with pip installations. Instead, use virtual environments or install packages with the --user flag for user-specific installations.
Package Conflicts When encountering package version conflicts, use virtual environments to isolate dependencies. Consider using tools like pipenv or conda for more advanced dependency management.

Conclusion

A properly configured Python environment is essential for productive development. Start with a clean Python installation, use virtual environments for project isolation, choose an appropriate IDE, and implement code quality tools. This foundation enables efficient development, easier collaboration, and maintainable codebases. Regular environment maintenance and staying updated with best practices will enhance your Python development experience.
========== Write your notes below==========

CHAPTER-3

Python Operators - Simple Guide

What are Operators?

Definition: Operators are special symbols that perform operations on variables and values. They are used to manipulate data and perform calculations, comparisons, and logical operations in Python programs.
Key Points:

- Operators work on operands (values or variables)
- Python has different types of operators for different purposes
- Operators have precedence rules that determine order of execution
- Some operators can work with multiple data types
- Results depend on the data types of operands

1. Arithmetic Operators

Definition: Arithmetic operators perform mathematical calculations on numeric values like addition, subtraction, multiplication, and division.
Important Points:

- Work with integers, floats, and complex numbers
- Division (/) always returns a float result
- Floor division (//) returns integer part only
- Modulus (%) returns remainder after division
- Exponentiation (**) raises number to a power

Operators and Examples:

```
a = 10
b = 3

# Addition
result = a + b    # 13

# Subtraction
result = a - b    # 7

# Multiplication
result = a * b    # 30
```

```
# Division
result = a / b    # 3.333...

# Floor Division
result = a // b   # 3

# Modulus (Remainder)
result = a % b    # 1

# Exponentiation
result = a ** b   # 1000
```

2. Comparison Operators

Definition: Comparison operators compare two values and return True or False based on the comparison result. They are used to make decisions in programs.
Important Points: • Always return boolean values (True or False) • Can compare numbers, strings, and other compatible types • String comparison is based on alphabetical order • Used in conditional statements and loops • Chaining comparisons is possible (a < b < c)
Operators and Examples:

```
x = 5
y = 10

# Equal to
result = x == y   # False

# Not equal to
result = x != y   # True

# Greater than
result = x > y    # False

# Less than
result = x < y    # True

# Greater than or equal to
result = x >= y   # False

# Less than or equal to
result = x <= y   # True
```

```
# String comparison
name1 = "Alice"
name2 = "Bob"
result = name1 < name2  # True (alphabetical order)
```

3. Logical Operators

Definition: Logical operators combine boolean values or expressions and return True or False. They are used to create complex conditions by combining multiple comparisons.
Important Points: • Work with boolean values (True/False) • 'and' returns True only if both conditions are True • 'or' returns True if at least one condition is True • 'not' reverses the boolean value • Use short-circuit evaluation for efficiency
Operators and Examples:

```
a = True
b = False
x = 5
y = 10

# AND operator
result = a and b       # False
result = (x < y) and (x > 0)  # True

# OR operator
result = a or b        # True
result = (x > y) or (x < 20)  # True

# NOT operator
result = not a         # False
result = not (x > y)    # True

# Combining logical operators
age = 25
has_license = True
result = (age >= 18) and has_license  # True
```

4. Assignment Operators

Definition: Assignment operators assign values to variables. They can perform an operation and assign the result back to the variable in a single step.

Important Points: • Basic assignment (=) assigns value to variable • Compound operators combine arithmetic and assignment • Modify the variable in place • Save writing time and make code more readable • Work with all numeric data types
Operators and Examples:

```
# Basic assignment
x = 10

# Addition assignment
x += 5    # Same as: x = x + 5, Result: 15

# Subtraction assignment
x -= 3    # Same as: x = x - 3, Result: 12

# Multiplication assignment
x *= 2    # Same as: x = x * 2, Result: 24

# Division assignment
x /= 4    # Same as: x = x / 4, Result: 6.0

# Floor division assignment
x //= 2   # Same as: x = x // 2, Result: 3.0

# Modulus assignment
x %= 2    # Same as: x = x % 2, Result: 1.0

# Exponentiation assignment
x **= 3   # Same as: x = x ** 3, Result: 1.0
```

5. Identity Operators

Definition: Identity operators check if two variables point to the same object in memory, not just if they have the same value.
Important Points: • 'is' checks if two variables refer to the same object • 'is not' checks if two variables refer to different objects • Different from equality (==) which checks values • Commonly used with None comparisons • Important for understanding Python's object model
Operators and Examples:

```
# is operator
a = [1, 2, 3]
b = [1, 2, 3]
c = a
```

```
result = a is b     # False (different objects)
result = a is c     # True (same object)

# is not operator
result = a is not b # True
result = a is not c # False

# Common use with None
value = None
result = value is None     # True
result = value is not None # False

# Integer caching example
x = 5
y = 5
result = x is y     # True (small integers are cached)
```

6. Membership Operators

Definition: Membership operators test if a value exists within a sequence (like strings, lists, tuples) or collection.
Important Points: • 'in' checks if value exists in the sequence • 'not in' checks if value does not exist in the sequence • Work with strings, lists, tuples, sets, and dictionaries • For dictionaries, checks for keys not values • Return boolean values (True/False)
Operators and Examples:

```
# in operator
text = "Hello World"
result = "Hello" in text      # True
result = "Python" in text     # False

numbers = [1, 2, 3, 4, 5]
result = 3 in numbers         # True
result = 6 in numbers         # False

# not in operator
result = "Python" not in text # True
result = 3 not in numbers     # False

# With dictionaries
student = {"name": "Alice", "age": 20}
```

```
result = "name" in student    # True (checks keys)
result = "Alice" in student   # False (not a key)

# With sets
colors = {"red", "green", "blue"}
result = "red" in colors      # True
```

7. Bitwise Operators

Definition: Bitwise operators perform operations on individual bits of integer numbers. They work at the binary level and are used for low-level programming tasks.
Important Points: • Work only with integer numbers • Operate on binary representation of numbers • Useful for system programming and optimization • & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift) • Results are integers
Operators and Examples:

```
a = 5   # Binary: 101
b = 3   # Binary: 011

# Bitwise AND
result = a & b    # 1 (Binary: 001)

# Bitwise OR
result = a | b    # 7 (Binary: 111)

# Bitwise XOR
result = a ^ b    # 6 (Binary: 110)

# Bitwise NOT
result = ~a       # -6

# Left shift
result = a << 1   # 10 (Binary: 1010)

# Right shift
result = a >> 1   # 2 (Binary: 10)
```

Operator Precedence

Definition: Operator precedence determines the order in which operations are performed when multiple operators are used in an expression.

Precedence Order (High to Low):

```
# Parentheses (highest)
result = (2 + 3) * 4       # 20

# Exponentiation
result = 2 ** 3 * 4        # 32

# Multiplication, Division, Modulus
result = 2 + 3 * 4         # 14

# Addition, Subtraction
result = 10 - 5 + 2        # 7

# Comparison operators
result = 5 > 3 and 2 < 4    # True

# Logical operators (lowest)
result = True or False and False  # True
```

Simple Examples

Combining Different Operators:

```
# Example 1: Simple calculator
num1 = 10
num2 = 5
sum_result = num1 + num2      # 15
is_greater = num1 > num2      # True

# Example 2: Age verification
age = 18
is_adult = age >= 18          # True
can_vote = is_adult and age >= 18  # True

# Example 3: String operations
name = "Python"
has_python = "Python" in name     # True
length_check = len(name) > 5      # True

# Example 4: List operations
numbers = [1, 2, 3, 4, 5]
has_three = 3 in numbers          # True
```

```
total = sum(numbers)            # 15
is_long_list = len(numbers) > 3   # True
```
========== Write your notes below==========

CHAPTER-4

Python Variables and Data Types - Detailed Guide

Variables in Python

Definition: A variable in Python is a named storage location that holds data values. It acts as a container or label that refers to a memory location where data is stored. Variables are created when you assign a value to them and can be reassigned to different values during program execution.
Key Points: • Variables are dynamically typed - no need to declare data type explicitly • Variable names are case-sensitive (age and Age are different variables) • Variables are created automatically when first assigned a value • Python uses reference-based assignment - variables point to objects in memory • Variable names must start with letter or underscore, followed by letters, digits, or underscores
Syntax:
variable_name = value

Examples:
# Basic variable assignment
name = "Alice"
age = 25
height = 5.6
is_student = True

Multiple assignment
x, y, z = 10, 20, 30

# Same value to multiple variables
a = b = c = 100

Python Data Types

Python has several built-in data types that can be categorized into different groups based on their characteristics and usage.

1. Numeric Data Types

Integer (int)

Definition: Integers are whole numbers without decimal points. They can be positive, negative, or zero. Python integers have unlimited precision, meaning they can be arbitrarily large.
Important Points: • No size limit - can store extremely large numbers • Supports binary (0b), octal (0o), and hexadecimal (0x) representations • Immutable data type - value cannot be changed after creation • Supports all arithmetic operations (+, -, *, /, //, %, **) • Can be created using int() constructor function
Syntax:
variable_name = integer_value

Examples:
```
# Basic integers
positive_num = 42
negative_num = -17
zero = 0

# Different number systems
binary_num = 0b1010    # Binary (equals 10 in decimal)
octal_num = 0o12       # Octal (equals 10 in decimal)
hex_num = 0xA          # Hexadecimal (equals 10 in decimal)

# Large integers
big_number = 12345678901234567890
```

Float (float)

Definition: Float represents real numbers with decimal points. They are implemented using double precision floating-point format and can represent both very large and very small numbers with fractional parts.
Important Points: • Limited precision due to floating-point representation (approximately 15-17 decimal digits) • Supports scientific notation using 'e' or 'E' (1.5e2 = 150.0) • Immutable data type like integers • Can result in precision errors during arithmetic operations • Special values: inf (infinity), -inf (negative infinity), nan (not a number)

Syntax:
variable_name = float_value

Examples:
```
# Basic floats
pi = 3.14159
temperature = -40.5
zero_float = 0.0

# Scientific notation
speed_of_light = 3.0e8    # 300,000,000.0
planck_constant = 6.626e-34

# Special values
positive_infinity = float('inf')
negative_infinity = float('-inf')
not_a_number = float('nan')
```

Complex (complex)

Definition: Complex numbers consist of a real part and an imaginary part. They are written in the form a+bj, where 'a' is the real part, 'b' is the imaginary part, and 'j' represents the imaginary unit (√-1).
Important Points: • Written as real + imaginary*j (j is the imaginary unit, not i) • Both real and imaginary parts are stored as floats • Supports all arithmetic operations and some mathematical functions • Immutable data type • Can be created using complex() constructor function
Syntax:
variable_name = real + imaginaryj
variable_name = complex(real, imaginary)

Examples:
```
# Basic complex numbers
z1 = 3 + 4j
z2 = 2 - 5j
z3 = 7j       # Pure imaginary number

# Using complex() function
z4 = complex(2, 3)    # 2 + 3j
z5 = complex(5)       # 5 + 0j

# Accessing real and imaginary parts
```

```
print(z1.real)      # 3.0
print(z1.imag)      # 4.0
```

2. Text Data Type

String (str)

Definition: Strings are sequences of characters enclosed in quotes. They represent textual data and are immutable, meaning once created, their content cannot be changed. Strings can contain letters, numbers, symbols, and special characters.
Important Points: • Immutable - cannot modify individual characters after creation • Can be created using single quotes (''), double quotes (""), or triple quotes (''' or """) • Supports indexing and slicing operations for accessing parts of the string • Rich set of built-in methods for string manipulation and processing • Unicode support - can contain characters from any language
Syntax:

```
variable_name = "string_value"
variable_name = 'string_value'
variable_name = """multiline_string"""
```

Examples:

```
# Different ways to create strings
single_quote = 'Hello World'
double_quote = "Python Programming"
triple_quote = """This is a
multiline string"""

# String indexing and slicing
text = "Python"
first_char = text[0]       # 'P'
last_char = text[-1]       # 'n'
substring = text[1:4]      # 'yth'

# String methods
message = "hello python"
upper_case = message.upper()       # "HELLO PYTHON"
capitalized = message.capitalize()  # "Hello python"
length = len(message)              # 12
```

3. Boolean Data Type

Boolean (bool)

Definition: Boolean represents logical values and can only have two possible values: True or False. It is used for logical operations, conditional statements, and control flow in programs.
Important Points: • Only two values: True and False (case-sensitive, must be capitalized) • Result of comparison operations (==, !=, <, >, <=, >=) • Used in conditional statements (if, while) and logical operations (and, or, not) • Any value can be evaluated as boolean using bool() function • Immutable data type
Syntax:
variable_name = True
variable_name = False

Examples:

```
# Direct boolean assignment
is_valid = True
is_complete = False

# Boolean from comparisons
age = 18
is_adult = age >= 18    # True
is_child = age < 13     # False

# Boolean operations
result1 = True and False    # False
result2 = True or False     # True
result3 = not True          # False

# Converting to boolean
bool(1)       # True
bool(0)       # False
bool("hello")  # True
bool("")      # False
```

4. Sequence Data Types

List (list)

Definition: A list is an ordered collection of items that can hold multiple values of different data types. Lists are mutable, meaning their contents can be modified after creation. Items are stored in a specific order and can be accessed using index positions.
Important Points: • Mutable - elements can be added, removed, or modified after creation • Ordered collection - maintains the sequence of elements as they were added • Allows duplicate elements and mixed data types in the same list • Supports indexing, slicing, and various built-in methods for manipulation • Created using square brackets [] or list() constructor
Syntax:
variable_name = [item1, item2, item3, ...]
variable_name = list([item1, item2, item3, ...])

Examples:
```
# Creating lists
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "hello", 3.14, True]
empty_list = []

# List operations
fruits = ["apple", "banana", "orange"]
fruits.append("grape")       # Add element
fruits.remove("banana")      # Remove element
fruits[0] = "mango"          # Modify element

# List slicing
subset = numbers[1:4]        # [2, 3, 4]
reversed_list = numbers[::-1] # [5, 4, 3, 2, 1]
```

Tuple (tuple)

Definition: A tuple is an ordered collection of items similar to a list, but it is immutable, meaning its contents cannot be changed after creation. Tuples are used to store multiple related values together as a single unit.
Important Points: • Immutable - cannot modify, add, or remove elements after creation • Ordered collection - maintains sequence and supports indexing • Allows duplicate elements and mixed data types • More memory efficient than lists for storing fixed data • Can be used as dictionary keys due to immutability
Syntax:

variable_name = (item1, item2, item3, ...)
variable_name = tuple([item1, item2, item3, ...])

Examples:
# Creating tuples
coordinates = (10, 20)
rgb_color = (255, 128, 0)
single_item = (42,) # Note the comma for single-item tuple

# Tuple operations
point = (3, 4, 5)
x, y, z = point # Tuple unpacking
length = len(point) # 3

# Accessing elements
first = point[0] # 3
last = point[-1] # 5
slice_tuple = point[1:3] # (4, 5)

5. Mapping Data Type

Dictionary (dict)

Definition: A dictionary is an unordered collection of key-value pairs where each key is unique and maps to a specific value. Dictionaries are mutable and provide fast lookup of values based on their associated keys.
Important Points: • Mutable - can add, remove, or modify key-value pairs after creation • Keys must be unique and immutable (strings, numbers, tuples) • Values can be of any data type and can be duplicated • Unordered collection (insertion order preserved from Python 3.7+) • Fast lookup time O(1) average case for accessing values by key
Syntax:
variable_name = {key1: value1, key2: value2, ...}
variable_name = dict(key1=value1, key2=value2, ...)

Examples:
# Creating dictionaries
student = {"name": "Alice", "age": 20, "grade": "A"}
empty_dict = {}

Dictionary operations
student["email"] = "alice@email.com" # Add new key-value pair

```
del student["grade"]                # Remove key-value pair
student["age"] = 21                 # Modify existing value

# Accessing dictionary elements
name = student["name"]          # "Alice"
age = student.get("age", 0)     # Safe access with default value
keys = student.keys()           # dict_keys(['name', 'age', 'email'])
values = student.values()       # dict_values(['Alice', 21, 'alice@email.com'])
```

6. Set Data Types

Set (set)

Definition: A set is an unordered collection of unique elements. Sets automatically eliminate duplicate values and are mutable, allowing addition and removal of elements. They are primarily used for mathematical set operations and removing duplicates.
Important Points: • Unordered collection - no indexing or slicing supported • Contains only unique elements - automatically removes duplicates • Mutable - can add or remove elements after creation • Elements must be immutable (hashable) types • Supports mathematical set operations (union, intersection, difference)
Syntax:

```
variable_name = {item1, item2, item3, ...}
variable_name = set([item1, item2, item3, ...])
```

Examples:

```
# Creating sets
numbers = {1, 2, 3, 4, 5}
mixed_set = {1, "hello", 3.14}
empty_set = set()    # Note: {} creates an empty dictionary

# Set operations
numbers.add(6)          # Add element
numbers.remove(1)       # Remove element
numbers.discard(10)     # Remove if exists (no error if not found)

# Set mathematical operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union = set1 | set2          # {1, 2, 3, 4, 5}
intersection = set1 & set2    # {3}
difference = set1 - set2      # {1, 2}
```

Type Checking and Conversion

Checking Data Types:

```
# Using type() function
x = 42
print(type(x))        # <class 'int'>

# Using isinstance() function
print(isinstance(x, int))    # True
print(isinstance(x, str))    # False
```

Type Conversion:

```
# Converting between types
int_val = int("42")        # String to integer
float_val = float("3.14")   # String to float
str_val = str(42)          # Integer to string
list_val = list("hello")    # String to list ['h', 'e', 'l', 'l', 'o']
tuple_val = tuple([1, 2, 3]) # List to tuple (1, 2, 3)
```

========== Write your notes below==========

CHAPTER-5

Python Conditional Statements - Simple Guide

What are Conditional Statements?

Definition: Conditional statements are programming constructs that allow you to execute different blocks of code based on whether certain conditions are true or false. They help programs make decisions and control the flow of execution.
Key Points: • Allow programs to make decisions based on conditions • Execute different code blocks depending on true/false evaluations • Use comparison and logical operators to create conditions • Essential for creating interactive and intelligent programs • Follow indentation rules in Python (4 spaces or 1 tab)

1. if Statement

Definition: The if statement executes a block of code only if the specified condition evaluates to True. It is the most basic form of conditional statement.
Important Points: • Executes code block only when condition is True • Condition must evaluate to a boolean value (True/False) • Code block must be indented (4 spaces recommended) • Colon (:) is required after the condition • If condition is False, the code block is skipped entirely
Syntax:

```
if condition:
    # code to execute if condition is True
```

Examples:

```
# Simple if statement
age = 18
if age >= 18:
    print("You are an adult")

# With multiple statements
score = 85
if score >= 80:
    print("Excellent work!")
    print("You passed with distinction")
```

```
# Using variables in condition
temperature = 30
if temperature > 25:
    print("It's a hot day")

# With boolean variable
is_raining = True
if is_raining:
    print("Take an umbrella")
```

2. if-else Statement

Definition: The if-else statement provides an alternative code block to execute when the if condition is False. It ensures that one of the two code blocks will always execute.
Important Points: • Provides two paths of execution - one for True, one for False • Exactly one block will always execute (either if or else) • else block executes when if condition is False • No condition needed for else block • Both blocks must be properly indented
Syntax:

```
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False
```

Examples:

```
# Basic if-else
age = 16
if age >= 18:
    print("You can vote")
else:
    print("You cannot vote yet")

# With calculations
number = 7
if number % 2 == 0:
    print("Number is even")
else:
    print("Number is odd")

# String comparison
password = "secret123"
```

```
if password == "admin123":
    print("Access granted")
else:
    print("Access denied")

# Boolean condition
has_ticket = False
if has_ticket:
    print("Welcome to the show")
else:
    print("Please buy a ticket first")
```

3. if-elif-else Statement

Definition: The if-elif-else statement allows you to check multiple conditions in sequence. It provides multiple alternative paths of execution, checking conditions one by one until a True condition is found.
Important Points: • Can have multiple elif blocks for different conditions • Conditions are checked in order from top to bottom • Only the first True condition's block executes • else block is optional and executes if no condition is True • More efficient than multiple separate if statements
Syntax:

```
if condition1:
    # code for condition1
elif condition2:
    # code for condition2
elif condition3:
    # code for condition3
else:
    # code if no condition is True
```

Examples:

```
# Grade classification
marks = 78
if marks >= 90:
    print("Grade: A+")
elif marks >= 80:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
elif marks >= 60:
```

```
    print("Grade: C")
else:
    print("Grade: F")

# Weather conditions
temperature = 22
if temperature > 30:
    print("Very hot day")
elif temperature > 20:
    print("Pleasant weather")
elif temperature > 10:
    print("Cool weather")
else:
    print("Cold day")

# Menu selection
choice = 2
if choice == 1:
    print("You selected Option 1")
elif choice == 2:
    print("You selected Option 2")
elif choice == 3:
    print("You selected Option 3")
else:
    print("Invalid choice")
```

4. Nested if Statements

Definition: Nested if statements are conditional statements placed inside other conditional statements. They allow you to create complex decision-making logic with multiple levels of conditions.

Important Points: • if statements can be placed inside other if, elif, or else blocks • Each level requires proper indentation (additional 4 spaces) • Inner conditions are checked only if outer conditions are True • Can create complex logic but should be kept readable • Alternative to using logical operators (and, or)

Syntax:

```
if outer_condition:
    if inner_condition:
        # code for both conditions True
    else:
        # code for outer True, inner False
```

```
else:
    # code for outer condition False

Examples:
# Age and license check
age = 20
has_license = True

if age >= 18:
    if has_license:
        print("You can drive")
    else:
        print("You need a license to drive")
else:
    print("You are too young to drive")

# Login system
username = "admin"
password = "pass123"

if username == "admin":
    if password == "pass123":
        print("Login successful")
    else:
        print("Wrong password")
else:
    print("User not found")

# Number analysis
number = 12
if number > 0:
    if number % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    if number == 0:
        print("Number is zero")
    else:
        print("Negative number")
```

5. Logical Operators in Conditions

Definition: Logical operators (and, or, not) can be used in conditional statements to combine multiple conditions and create more complex decision-making logic.
Important Points: • 'and' requires all conditions to be True • 'or' requires at least one condition to be True • 'not' reverses the boolean result • Can replace some nested if statements for cleaner code • Use parentheses for complex expressions to ensure correct order
Examples:

```
# Using 'and' operator
age = 25
salary = 50000
if age >= 18 and salary >= 30000:
    print("Loan approved")

# Using 'or' operator
day = "Saturday"
if day == "Saturday" or day == "Sunday":
    print("It's weekend!")

# Using 'not' operator
is_logged_in = False
if not is_logged_in:
    print("Please log in first")

# Combining multiple logical operators
score = 85
attendance = 90
if score >= 80 and attendance >= 85:
    print("You get a certificate")
elif score >= 70 or attendance >= 90:
    print("You pass the course")
else:
    print("You need to improve")

# Complex condition with parentheses
age = 22
income = 25000
has_job = True
if (age >= 21 and income >= 20000) or has_job:
    print("Credit card approved")
```

6. Ternary Operator (Conditional Expression)

Definition: The ternary operator is a shorthand way to write simple if-else statements in a single line. It returns one value if the condition is True and another value if False.
Important Points: • Provides a concise way to write simple if-else logic • Returns a value that can be assigned to a variable • Best used for simple conditions and short expressions • Makes code more readable for simple decisions • Also called conditional expression
Syntax:
variable = value_if_true if condition else value_if_false

Examples:

```
# Basic ternary operator
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)  # Adult

# With calculations
a = 10
b = 5
max_value = a if a > b else b
print(max_value)  # 10

# In print statements
score = 75
print("Pass" if score >= 60 else "Fail")

# Multiple ternary operators
number = -5
result = "Positive" if number > 0 else ("Zero" if number == 0 else "Negative")
print(result)  # Negative

# With function calls
def get_discount(is_member):
    return 20 if is_member else 5

discount = get_discount(True)
print(f"Your discount: {discount}%")  # Your discount: 20%
```

Common Examples and Use Cases

Example 1: Simple Calculator

```
operation = "+"
num1 = 10
num2 = 5

if operation == "+":
    result = num1 + num2
elif operation == "-":
    result = num1 - num2
elif operation == "*":
    result = num1 * num2
elif operation == "/":
    result = num1 / num2
else:
    result = "Invalid operation"

print(f"Result: {result}")

Example 2: Student Grade System
marks = 88
attendance = 85

if marks >= 90 and attendance >= 90:
    grade = "A+"
elif marks >= 80 and attendance >= 80:
    grade = "A"
elif marks >= 70:
    grade = "B"
elif marks >= 60:
    grade = "C"
else:
    grade = "F"

print(f"Your grade: {grade}")

Example 3: User Authentication
username = "john"
password = "secret"
is_active = True

if username == "john":
    if password == "secret":
        if is_active:
```

```
            print("Login successful")
        else:
            print("Account is deactivated")
    else:
        print("Wrong password")
else:
    print("User not found")
```

Example 4: Shopping Discount System

```
total_amount = 150
is_member = True
is_first_time = False

if total_amount >= 100:
    if is_member:
        discount = 15
    elif is_first_time:
        discount = 10
    else:
        discount = 5
else:
    discount = 0

final_amount = total_amount - (total_amount * discount / 100)
print(f"Discount: {discount}%")
print(f"Final amount: ${final_amount}")
```

CHAPTER-6

Python Loops - Simple Guide

What are Loops?

Definition: Loops are programming constructs that allow you to execute a block of code repeatedly until a specific condition is met. They help automate repetitive tasks and iterate through data collections efficiently.

Key Points: • Execute code blocks multiple times without rewriting the same code • Control repetition using conditions or counting mechanisms • Essential for processing collections like lists, strings, and dictionaries • Help reduce code duplication and improve program efficiency • Two main types: for loops (definite iteration) and while loops (indefinite iteration)

1. for Loop

Definition: A for loop iterates over a sequence (like a list, tuple, string, or range) and executes a block of code for each element in that sequence. It's used when you know in advance how many times you want to repeat something.
Important Points: • Iterates through each element of a sequence automatically • Loop variable takes the value of each element in turn • Commonly used with range() function for counting loops • Automatically stops when sequence is exhausted • Most efficient way to iterate through collections
Syntax:

```
for variable in sequence:
    # code to execute for each element
```

Examples:
Basic for loop with list:

```
# Iterating through a list
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)
# Output: apple, banana, orange (each on new line)

# Iterating through numbers
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num * 2)
# Output: 2, 4, 6, 8, 10
```

Using range() function:

```
# Simple counting loop
for i in range(5):
    print(f"Count: {i}")
# Output: Count: 0, Count: 1, Count: 2, Count: 3, Count: 4

# Range with start and stop
for i in range(1, 6):
    print(i)
```

```
# Output: 1, 2, 3, 4, 5

# Range with step
for i in range(0, 10, 2):
    print(i)
# Output: 0, 2, 4, 6, 8
```

String iteration:

```
# Iterating through characters
word = "Python"
for letter in word:
    print(letter)
# Output: P, y, t, h, o, n (each on new line)

# Counting characters
text = "Hello"
count = 0
for char in text:
    count += 1
print(f"Length: {count}")  # Length: 5
```

2. while Loop

Definition: A while loop continues to execute a block of code as long as a specified condition remains True. It's used when you don't know exactly how many times you need to repeat something.

Important Points: • Continues execution while condition is True • Condition is checked before each iteration • Must update loop variable inside the loop to avoid infinite loops • Used when number of iterations is not predetermined • Stops immediately when condition becomes False

Syntax:

```
while condition:
    # code to execute while condition is True
    # update condition variable
```

Examples:

Basic counting with while:

```
# Simple counting
count = 1
while count <= 5:
    print(f"Count: {count}")
```

```
    count += 1
# Output: Count: 1, Count: 2, Count: 3, Count: 4, Count: 5

# Countdown
num = 5
while num > 0:
    print(num)
    num -= 1
print("Blast off!")
# Output: 5, 4, 3, 2, 1, Blast off!
```

User input validation:

```
# Keep asking until valid input
password = ""
while password != "secret":
    password = input("Enter password: ")
    if password != "secret":
        print("Wrong password, try again")
print("Access granted!")

# Number guessing
target = 7
guess = 0
while guess != target:
    guess = int(input("Guess the number (1-10): "))
    if guess != target:
        print("Try again!")
print("Correct!")
```

3. Nested Loops

Definition: Nested loops are loops placed inside other loops. The inner loop completes all its iterations for each iteration of the outer loop, creating a pattern of repeated execution.

Important Points: • Inner loop executes completely for each iteration of outer loop • Useful for working with 2D data structures like matrices • Can create complex patterns and process multi-dimensional data • Each loop can be either for or while loop • Indentation is crucial for proper nesting

Examples:

Basic nested for loops:

```
# Multiplication table
```

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j}")
    print()  # Empty line after each table

# Pattern printing
for i in range(3):
    for j in range(3):
        print("*", end=" ")
    print()  # New line after each row
# Output:
# * * *
# * * *
# * * *
```

Matrix processing:

```
# 2D list (matrix)
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for element in row:
        print(element, end=" ")
    print()  # New line after each row

# Sum of all elements
total = 0
for row in matrix:
    for element in row:
        total += element
print(f"Sum: {total}")  # Sum: 45
```

4. Loop Control Statements

break Statement

Definition: The break statement immediately terminates the loop and transfers control to the statement after the loop. It's used to exit a loop prematurely when a certain condition is met.

Important Points: • Immediately exits the current loop • In nested loops, only exits the innermost loop • Often used with conditional statements • Transfers control to first statement after the loop • Commonly used to stop infinite loops or early termination

Examples:

```
# Break in for loop
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if num == 5:
        break
    print(num)
# Output: 1, 2, 3, 4

# Break in while loop
count = 1
while True:  # Infinite loop
    print(count)
    count += 1
    if count > 3:
        break
# Output: 1, 2, 3

# Finding first even number
numbers = [1, 3, 5, 8, 9, 10]
for num in numbers:
    if num % 2 == 0:
        print(f"First even number: {num}")
        break
```

continue Statement

Definition: The continue statement skips the rest of the current iteration and moves to the next iteration of the loop. It doesn't terminate the loop but jumps to the beginning of the next cycle.
Important Points: • Skips remaining statements in current iteration • Moves directly to next iteration • Loop continues normally after continue • Useful for skipping unwanted values • Often used with conditional statements
Examples:

```
# Skip even numbers
for i in range(1, 11):
    if i % 2 == 0:
        continue
    print(i)
# Output: 1, 3, 5, 7, 9

# Skip specific values
numbers = [1, 2, 3, 4, 5]
```

```
for num in numbers:
    if num == 3:
        continue
    print(num * 2)
# Output: 2, 4, 8, 10 (skips 3*2=6)

# Process only positive numbers
values = [-2, 3, -1, 5, 0, 7]
for val in values:
    if val <= 0:
        continue
    print(f"Positive: {val}")
# Output: Positive: 3, Positive: 5, Positive: 7
```

pass Statement

Definition: The pass statement is a null operation - it does nothing when executed. It's used as a placeholder where syntactically some code is required but no action needs to be taken.
Important Points: • Does nothing - null operation • Used as placeholder for future code • Maintains code structure during development • Required where Python expects an indented block • Commonly used in empty loops, functions, or classes
Examples:

```
# Placeholder for future development
for i in range(5):
    if i == 2:
        pass  # TODO: Add special handling for 2
    else:
        print(i)

# Empty loop that does nothing
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    pass  # Loop runs but does nothing

# Conditional placeholder
x = 10
if x > 5:
    pass  # Will implement later
else:
    print("Small number")
```

5. Loop with else Clause

Definition: Python loops can have an optional else clause that executes when the loop completes normally (without encountering a break statement). It provides a way to execute cleanup code or handle the case when loop finishes all iterations.
Important Points: • Executes only when loop completes normally (not broken) • Does not execute if loop is terminated by break statement • Useful for search operations and cleanup tasks • Works with both for and while loops • else block is optional
Examples:

```
# Search with for loop
numbers = [1, 3, 5, 7, 9]
target = 4
for num in numbers:
    if num == target:
        print(f"Found {target}")
        break
else:
    print(f"{target} not found in list")
# Output: 4 not found in list

# While loop with else
count = 1
while count <= 3:
    print(count)
    count += 1
else:
    print("Loop completed normally")
# Output: 1, 2, 3, Loop completed normally

# Password attempts
attempts = 0
max_attempts = 3
while attempts < max_attempts:
    password = input("Enter password: ")
    if password == "secret":
        print("Login successful")
        break
    attempts += 1
else:
    print("Too many failed attempts")
```

Common Loop Patterns and Examples

Example 1: Sum of numbers

```
# Using for loop
numbers = [1, 2, 3, 4, 5]
total = 0
for num in numbers:
    total += num
print(f"Sum: {total}")  # Sum: 15

# Using while loop
total = 0
i = 1
while i <= 5:
    total += i
    i += 1
print(f"Sum: {total}")  # Sum: 15
```

Example 2: Finding maximum

```
numbers = [23, 45, 12, 67, 34]
max_num = numbers[0]
for num in numbers:
    if num > max_num:
        max_num = num
print(f"Maximum: {max_num}")  # Maximum: 67
```

Example 3: Counting occurrences

```
text = "hello world"
letter = "l"
count = 0
for char in text:
    if char == letter:
        count += 1
print(f"'{letter}' appears {count} times")  # 'l' appears 3 times
```

Example 4: List processing

```
# Square all numbers
numbers = [1, 2, 3, 4, 5]
squares = []
for num in numbers:
    squares.append(num ** 2)
print(squares)  # [1, 4, 9, 16, 25]
```

```
# Filter even numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
print(even_numbers)  # [2, 4, 6, 8, 10]

Example 5: Menu system
while True:
    print("\n--- Menu ---")
    print("1. Add")
    print("2. Subtract")
    print("3. Exit")

    choice = input("Enter choice: ")

    if choice == "1":
        print("Addition selected")
    elif choice == "2":
        print("Subtraction selected")
    elif choice == "3":
        print("Goodbye!")
        break
    else:
        print("Invalid choice")
```

CHAPTER-7

Python Data Structures - Lists, Tuples, Dictionaries, Sets

What are Data Structures?

Definition: Data structures are ways of organizing and storing data in computer memory so that it can be accessed and used efficiently. Python provides built-in data structures that help manage collections of related data.

Key Points: • Organize and store multiple pieces of data in a single variable • Each data structure has unique characteristics and use cases • Choose the right data structure based on your needs (mutability, ordering, duplicates) • Essential for efficient data management and algorithm implementation • Built-in data structures are optimized for performance and ease of use

1. Lists

Definition: A list is an ordered, mutable collection that can store multiple items of different data types. Lists maintain the order of elements and allow duplicates, making them versatile for various programming tasks.
Important Points: • Mutable - elements can be changed, added, or removed after creation • Ordered - maintains the sequence of elements as they were added • Allows duplicate elements and mixed data types • Uses square brackets [] for creation and indexing for access • Dynamic size - can grow or shrink during program execution
Syntax:
list_name = [item1, item2, item3, ...]
list_name = list([item1, item2, item3, ...])

Creating Lists

```
# Empty list
empty_list = []
numbers = list()

# List with initial values
fruits = ["apple", "banana", "orange"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]

# List from range
nums = list(range(1, 6))  # [1, 2, 3, 4, 5]
```

Accessing List Elements

```
fruits = ["apple", "banana", "orange", "grape"]

# Positive indexing (starts from 0)
first_fruit = fruits[0]    # "apple"
second_fruit = fruits[1]   # "banana"
```

```
# Negative indexing (starts from -1)
last_fruit = fruits[-1]    # "grape"
second_last = fruits[-2]   # "orange"

# Slicing
subset = fruits[1:3]      # ["banana", "orange"]
first_two = fruits[:2]    # ["apple", "banana"]
last_two = fruits[-2:]    # ["orange", "grape"]
```

Modifying Lists

```
fruits = ["apple", "banana", "orange"]

# Changing elements
fruits[0] = "mango"       # ["mango", "banana", "orange"]

# Adding elements
fruits.append("grape")    # Add at end
fruits.insert(1, "kiwi")  # Insert at index 1

# Removing elements
fruits.remove("banana")   # Remove first occurrence
last_item = fruits.pop()  # Remove and return last item
del fruits[0]             # Delete by index

# Extending lists
more_fruits = ["pear", "peach"]
fruits.extend(more_fruits) # Add all elements
```

Common List Methods

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

# Information methods
length = len(numbers)     # 8
count_ones = numbers.count(1)  # 2
index_of_4 = numbers.index(4)  # 2

# Sorting and reversing
numbers.sort()            # Sort in place: [1, 1, 2, 3, 4, 5, 6, 9]
numbers.reverse()         # Reverse in place
```

```
sorted_copy = sorted(numbers)  # Create sorted copy

# Other useful methods
numbers.clear()          # Remove all elements
copy_list = numbers.copy() # Create shallow copy
```

2. Tuples

Definition: A tuple is an ordered, immutable collection that can store multiple items of different data types. Once created, tuple elements cannot be changed, making them useful for storing fixed data.
Important Points: • Immutable - elements cannot be changed after creation • Ordered - maintains sequence and supports indexing • Allows duplicate elements and mixed data types • Uses parentheses () for creation (optional for simple cases) • More memory efficient than lists for storing fixed data
Syntax:

```
tuple_name = (item1, item2, item3, ...)
tuple_name = tuple([item1, item2, item3, ...])
```

Creating Tuples

```
# Empty tuple
empty_tuple = ()
empty_tuple2 = tuple()

# Tuple with values
coordinates = (10, 20)
colors = ("red", "green", "blue")
mixed = (1, "hello", 3.14, True)

# Single element tuple (comma required)
single = (42,)
single2 = 42,

# Without parentheses (tuple packing)
point = 3, 4, 5
```

Accessing Tuple Elements

```
student = ("Alice", 20, "Computer Science", 3.8)
```

```
# Indexing (same as lists)
name = student[0]        # "Alice"
age = student[1]         # 20
last_item = student[-1]   # 3.8

# Slicing
info = student[1:3]       # (20, "Computer Science")

# Tuple unpacking
name, age, major, gpa = student
print(f"{name} is {age} years old")
```

Tuple Methods and Operations

```
numbers = (1, 2, 3, 2, 4, 2, 5)

# Information methods
length = len(numbers)     # 7
count_twos = numbers.count(2)  # 3
index_of_4 = numbers.index(4)  # 4

# Concatenation (creates new tuple)
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined = tuple1 + tuple2  # (1, 2, 3, 4, 5, 6)

# Repetition
repeated = (1, 2) * 3     # (1, 2, 1, 2, 1, 2)

# Membership testing
has_three = 3 in numbers   # True
```

When to Use Tuples

```
# Coordinates
point = (10, 20)
rgb_color = (255, 128, 0)

# Database records
employee = ("John Doe", 101, "Manager", 75000)
```

```
# Function returning multiple values
def get_name_age():
    return "Alice", 25

name, age = get_name_age()

# Dictionary keys (immutable required)
locations = {(0, 0): "origin", (1, 1): "diagonal"}
```

3. Dictionaries

Definition: A dictionary is an unordered collection of key-value pairs where each key is unique and maps to a specific value. Dictionaries provide fast lookup and are perfect for storing related data.
Important Points: • Mutable - can add, remove, or modify key-value pairs • Keys must be unique and immutable (strings, numbers, tuples) • Values can be any data type and can be duplicated • Unordered collection (insertion order preserved from Python 3.7+) • Fast lookup time O(1) for accessing values by key
Syntax:
dict_name = {key1: value1, key2: value2, ...}
dict_name = dict(key1=value1, key2=value2, ...)

Creating Dictionaries

```
# Empty dictionary
empty_dict = {}
empty_dict2 = dict()

# Dictionary with initial values
student = {"name": "Alice", "age": 20, "grade": "A"}
scores = {1: 85, 2: 92, 3: 78}

# Using dict() constructor
person = dict(name="Bob", age=25, city="New York")

# From list of tuples
items = [("a", 1), ("b", 2), ("c", 3)]
letter_dict = dict(items)  # {"a": 1, "b": 2, "c": 3}
```

Accessing Dictionary Elements

```
student = {"name": "Alice", "age": 20, "major": "CS", "gpa": 3.8}

# Direct access (raises KeyError if key doesn't exist)
name = student["name"]    # "Alice"
age = student["age"]      # 20

# Safe access with get() method
gpa = student.get("gpa")   # 3.8
phone = student.get("phone", "Not available")  # "Not available"

# Check if key exists
has_name = "name" in student    # True
has_phone = "phone" in student  # False
```

Modifying Dictionaries

```
student = {"name": "Alice", "age": 20}

# Adding new key-value pairs
student["major"] = "Computer Science"
student["gpa"] = 3.8

# Modifying existing values
student["age"] = 21

# Removing elements
del student["gpa"]        # Remove specific key
major = student.pop("major", "Unknown")  # Remove and return value
last_item = student.popitem()  # Remove and return last item

# Updating with another dictionary
new_info = {"city": "Boston", "year": 2}
student.update(new_info)
```

Dictionary Methods and Operations

```
student = {"name": "Alice", "age": 20, "major": "CS"}

# Getting keys, values, and items
keys = student.keys()     # dict_keys(['name', 'age', 'major'])
```

```
values = student.values()  # dict_values(['Alice', 20, 'CS'])
items = student.items()    # dict_items([('name', 'Alice'), ...])

# Converting to lists
key_list = list(student.keys())
value_list = list(student.values())

# Copying
student_copy = student.copy()

# Clearing all elements
student.clear()

# Creating from keys
keys = ["a", "b", "c"]
default_dict = dict.fromkeys(keys, 0)  # {"a": 0, "b": 0, "c": 0}
```

Dictionary Iteration

```
grades = {"Alice": 85, "Bob": 92, "Charlie": 78}

# Iterate over keys
for student in grades:
    print(student)

# Iterate over values
for grade in grades.values():
    print(grade)

# Iterate over key-value pairs
for student, grade in grades.items():
    print(f"{student}: {grade}")
```

4. Sets

Definition: A set is an unordered collection of unique elements. Sets automatically eliminate duplicates and are primarily used for mathematical set operations and removing duplicates from data.
Important Points: • Unordered collection - no indexing or slicing supported • Contains only unique elements - automatically removes duplicates • Mutable - can add or remove

elements after creation • Elements must be immutable (hashable) types • Supports mathematical set operations (union, intersection, difference)
Syntax:
set_name = {item1, item2, item3, ...}
set_name = set([item1, item2, item3, ...])

Creating Sets

```
# Empty set (must use set() constructor)
empty_set = set()

# Set with initial values
fruits = {"apple", "banana", "orange"}
numbers = {1, 2, 3, 4, 5}

# From list (removes duplicates)
list_with_dupes = [1, 2, 2, 3, 3, 3, 4]
unique_numbers = set(list_with_dupes)  # {1, 2, 3, 4}

# From string
letters = set("hello")     # {'h', 'e', 'l', 'o'}
```

Modifying Sets

```
fruits = {"apple", "banana"}

# Adding elements
fruits.add("orange")      # Add single element
fruits.update(["grape", "kiwi"])  # Add multiple elements

# Removing elements
fruits.remove("banana")    # Raises KeyError if not found
fruits.discard("mango")    # No error if not found
removed = fruits.pop()     # Remove and return arbitrary element

# Clearing all elements
fruits.clear()
```

Set Operations

```
set1 = {1, 2, 3, 4}
```

```
set2 = {3, 4, 5, 6}

# Union (all elements from both sets)
union = set1 | set2       # {1, 2, 3, 4, 5, 6}
union2 = set1.union(set2)

# Intersection (common elements)
intersection = set1 & set2  # {3, 4}
intersection2 = set1.intersection(set2)

# Difference (elements in set1 but not set2)
difference = set1 - set2   # {1, 2}
difference2 = set1.difference(set2)

# Symmetric difference (elements in either set, but not both)
sym_diff = set1 ^ set2     # {1, 2, 5, 6}
sym_diff2 = set1.symmetric_difference(set2)
```

Set Methods and Testing

```
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}

# Membership testing
has_two = 2 in set1       # True
has_six = 6 in set1       # False

# Subset and superset testing
is_subset = set1.issubset(set2)    # True
is_superset = set2.issuperset(set1) # True
is_disjoint = set1.isdisjoint({4, 5, 6})  # True

# Set size
length = len(set1)        # 3

# Copy set
set_copy = set1.copy()
```

Choosing the Right Data Structure

Comparison Table

```
# When to use each data structure:

# Lists - When you need:
# ✓ Ordered collection with duplicates
# ✓ Mutable sequence for frequent changes
# ✓ Indexing and slicing capabilities
shopping_list = ["milk", "bread", "eggs", "milk"]

# Tuples - When you need:
# ✓ Ordered collection that won't change
# ✓ Data integrity (immutable)
# ✓ Dictionary keys or set elements
coordinates = (10, 20)

# Dictionaries - When you need:
# ✓ Key-value relationships
# ✓ Fast lookups by unique keys
# ✓ Structured data storage
student_info = {"name": "Alice", "grade": 85}

# Sets - When you need:
# ✓ Unique elements only
# ✓ Mathematical set operations
# ✓ Fast membership testing
unique_visitors = {"user1", "user2", "user3"}
```

Practical Examples

Example 1: Contact Management

```
# Using dictionary for contact storage
contacts = {
    "Alice": {"phone": "123-456-7890", "email": "alice@email.com"},
    "Bob": {"phone": "987-654-3210", "email": "bob@email.com"}
}

# Adding new contact
```

```
contacts["Charlie"] = {"phone": "555-123-4567", "email": "charlie@email.com"}

# Accessing contact info
alice_phone = contacts["Alice"]["phone"]
```

Example 2: Student Grade Management

```
# List of students (ordered, with possible duplicates)
students = ["Alice", "Bob", "Charlie", "Alice"]

# Dictionary for grades
grades = {"Alice": [85, 92, 78], "Bob": [90, 88, 95], "Charlie": [82, 79, 91]}

# Set for unique subjects
subjects = {"Math", "Science", "English", "History"}

# Tuple for student record (immutable)
student_record = ("Alice", 20, "Computer Science", 3.8)
```

Example 3: Data Processing

```
# Remove duplicates from list using set
data = [1, 2, 2, 3, 4, 4, 5]
unique_data = list(set(data))  # [1, 2, 3, 4, 5]

# Count occurrences using dictionary
text = "hello world"
char_count = {}
for char in text:
    char_count[char] = char_count.get(char, 0) + 1

# Find common elements between lists
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
common = list(set(list1) & set(list2))  # [4, 5]
```

Example 4: Inventory Management

```
# Dictionary for inventory
inventory = {
    "apples": {"quantity": 50, "price": 0.5},
```

```
    "bananas": {"quantity": 30, "price": 0.3},
    "oranges": {"quantity": 25, "price": 0.6}
}

# List for shopping cart (allows duplicates)
cart = ["apples", "bananas", "apples", "oranges"]

# Set for available categories
categories = {"fruits", "vegetables", "dairy", "meat"}

# Tuple for product info (immutable)
product_info = ("apples", "Red Delicious", "Fresh", 0.5)
==================== Write your notes below=========================
```

CHAPTER-8

Python Functions - Simple Guide

What are Functions?

Definition: A function is a reusable block of organized code that performs a specific task. Functions take input (parameters), process it, and return output (result). They help break down complex problems into smaller, manageable pieces.
Key Points: • Reusable code blocks that avoid repetition and reduce errors • Take input through parameters and return results using return statement • Help organize code into logical, manageable sections • Can be called multiple times from different parts of the program • Follow DRY principle (Don't Repeat Yourself) for cleaner code

1. Function Definition and Syntax

Definition: Function definition creates a named block of code that can be executed later. It specifies what the function does, what parameters it accepts, and what it returns.

Important Points: • Defined using 'def' keyword followed by function name and parentheses • Function name should be descriptive and follow snake_case convention • Colon (:) is required after the parameter list • Function body must be indented (4 spaces recommended) • Functions must be defined before they are called
Syntax:

```
def function_name(parameters):
    """Optional docstring"""
    # function body
    return value  # optional
```

Examples:

```
# Simple function without parameters
def greet():
    print("Hello, World!")

# Function with parameters
def greet_person(name):
    print(f"Hello, {name}!")

# Function with return value
def add_numbers(a, b):
    result = a + b
    return result

# Function with docstring
def calculate_area(length, width):
    """Calculate area of rectangle"""
    area = length * width
    return area
```

2. Function Parameters and Arguments

Definition: Parameters are variables defined in the function definition that receive values when the function is called. Arguments are the actual values passed to the function when calling it.
Important Points: • Parameters are placeholders in function definition • Arguments are actual values passed during function call • Parameters act as local variables inside the function • Number of arguments must match number of parameters (unless defaults used) • Parameters can have default values for optional arguments
Examples:
Basic parameters:

```
# Function with single parameter
def square(number):
    return number * number

result = square(5)  # 5 is the argument
print(result)  # 25

# Function with multiple parameters
def introduce(name, age, city):
    print(f"My name is {name}, I am {age} years old, from {city}")

introduce("Alice", 25, "New York")
```

Default parameters:

```
# Function with default parameter
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")  # Uses default greeting
greet("Bob", "Hi")  # Uses custom greeting

# Multiple default parameters
def create_profile(name, age=18, country="USA"):
    return f"Name: {name}, Age: {age}, Country: {country}"

profile1 = create_profile("John")
profile2 = create_profile("Jane", 25)
profile3 = create_profile("Mike", 30, "Canada")
```

3. Return Statement

Definition: The return statement sends a result back to the caller and immediately exits the function. It can return any type of value or nothing at all (None).
Important Points: • Immediately exits the function when executed • Can return any data type (numbers, strings, lists, etc.) • Function returns None if no return statement is used • Can have multiple return statements in different conditions • Only one return value is sent back per function call
Examples:

```
# Function returning a number
def multiply(a, b):
    return a * b
```

```
result = multiply(4, 5)  # result = 20

# Function returning a string
def get_full_name(first, last):
    return first + " " + last

name = get_full_name("John", "Doe")  # name = "John Doe"

# Function returning multiple values (tuple)
def get_name_age():
    return "Alice", 25

name, age = get_name_age()  # Tuple unpacking

# Conditional return
def check_even_odd(number):
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"

# Function without return (returns None)
def print_message(msg):
    print(msg)
    # No return statement - returns None

result = print_message("Hello")  # result is None
```

4. Local and Global Variables

Definition: Local variables are created inside functions and can only be accessed within that function. Global variables are created outside functions and can be accessed from anywhere in the program.

Important Points: • Local variables exist only during function execution • Global variables persist throughout program execution • Local variables have higher priority than global variables with same name • Use 'global' keyword to modify global variables inside functions • Good practice to avoid global variables when possible

Examples:

```
# Global variable
counter = 0
```

```
def increment():
    # Local variable
    local_counter = 1
    print(f"Local counter: {local_counter}")

def increment_global():
    global counter
    counter += 1
    print(f"Global counter: {counter}")

# Function with local variable shadowing global
name = "Global Name"

def show_names():
    name = "Local Name"  # Shadows global variable
    print(f"Inside function: {name}")

show_names()  # Inside function: Local Name
print(f"Outside function: {name}")  # Outside function: Global Name

# Reading global variable (no global keyword needed)
total = 100

def show_total():
    print(f"Total is: {total}")  # Can read global variable

show_total()  # Total is: 100
```

5. Types of Arguments

Positional Arguments

Definition: Positional arguments are passed to functions in a specific order, and their position determines which parameter receives each value.
Examples:

```
def describe_pet(name, animal_type, age):
    print(f"Pet name: {name}")
    print(f"Animal type: {animal_type}")
    print(f"Age: {age}")
```

```
# Positional arguments - order matters
describe_pet("Buddy", "Dog", 3)
```

Keyword Arguments

Definition: Keyword arguments are passed using parameter names, allowing arguments to be passed in any order.
Examples:

```
# Same function, different call style
describe_pet(animal_type="Cat", name="Whiskers", age=2)
describe_pet(name="Max", age=5, animal_type="Dog")

# Mixing positional and keyword arguments
describe_pet("Luna", animal_type="Cat", age=1)
```

Variable-Length Arguments

Definition: Functions can accept varying numbers of arguments using *args (for positional) and **kwargs (for keyword arguments).
Important Points: • *args collects extra positional arguments into a tuple • **kwargs collects extra keyword arguments into a dictionary • Must come after regular parameters in function definition • Can be used together in the same function • Common convention to use 'args' and 'kwargs' as names
Examples:

```
# *args for variable positional arguments
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

result1 = sum_all(1, 2, 3)  # 6
result2 = sum_all(1, 2, 3, 4, 5)  # 15

# **kwargs for variable keyword arguments
def create_person(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

create_person(name="John", age=30, city="New York")
```

```
# Combining regular, *args, and **kwargs
def process_data(required_param, *args, **kwargs):
    print(f"Required: {required_param}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

process_data("Must have", 1, 2, 3, name="Test", value=100)
```

6. Lambda Functions (Anonymous Functions)

Definition: Lambda functions are small, anonymous functions defined using the lambda keyword. They can have multiple parameters but only one expression, and they automatically return the result.
Important Points: • Single-line functions without names • Can have multiple parameters but only one expression • Automatically return the expression result • Commonly used with functions like map(), filter(), sort() • Good for simple operations, not complex logic
Syntax:
lambda parameters: expression

Examples:

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

# Lambda with multiple parameters
add = lambda a, b: a + b
print(add(3, 4))  # 7

# Using lambda with built-in functions
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))
print(squares)  # [1, 4, 9, 16, 25]

# Using lambda with filter
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4, 6, 8, 10]

# Lambda in sorting
students = [("Alice", 85), ("Bob", 90), ("Charlie", 78)]
students.sort(key=lambda student: student[1])  # Sort by grade
```

```
print(students)  # [('Charlie', 78), ('Alice', 85), ('Bob', 90)]
```

7. Built-in Functions

Definition: Python provides many built-in functions that perform common tasks without needing to be defined. These functions are always available and cover basic operations.
Important Points: • Always available without importing modules • Cover common programming tasks and operations • Well-tested and optimized for performance • Include functions for type conversion, math, input/output • Help reduce code complexity and development time
Common Built-in Functions:

```
# Type conversion functions
num_str = "123"
num_int = int(num_str)  # 123
num_float = float(num_str)  # 123.0
back_to_str = str(num_int)  # "123"

# Math functions
numbers = [1, 5, 3, 9, 2]
maximum = max(numbers)  # 9
minimum = min(numbers)  # 1
total = sum(numbers)    # 20
length = len(numbers)   # 5
absolute = abs(-10)     # 10

# Input/Output
name = input("Enter your name: ")
print(f"Hello, {name}!")

# Iteration functions
for i, value in enumerate(['a', 'b', 'c']):
    print(f"Index {i}: {value}")

# Range function
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4

# Type checking
print(type(123))       # <class 'int'>
print(isinstance(123, int))  # True
```

Common Function Examples

Example 1: Calculator Functions

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

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

def multiply(a, b):
    return a * b

def divide(a, b):
    if b != 0:
        return a / b
    else:
        return "Cannot divide by zero"

# Using the functions
result1 = add(10, 5)       # 15
result2 = multiply(4, 3)    # 12
result3 = divide(10, 2)     # 5.0
```

Example 2: String Processing Functions

```
def capitalize_words(text):
    words = text.split()
    capitalized = []
    for word in words:
        capitalized.append(word.capitalize())
    return " ".join(capitalized)

def count_vowels(text):
    vowels = "aeiouAEIOU"
    count = 0
    for char in text:
        if char in vowels:
            count += 1
    return count

# Using the functions
sentence = "hello world"
result1 = capitalize_words(sentence)  # "Hello World"
```

```
result2 = count_vowels(sentence)     # 3

Example 3: List Processing Functions
def find_maximum(numbers):
    if not numbers:
        return None
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num

def filter_even(numbers):
    even_numbers = []
    for num in numbers:
        if num % 2 == 0:
            even_numbers.append(num)
    return even_numbers

def calculate_average(numbers):
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)

# Using the functions
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
max_val = find_maximum(data)     # 10
evens = filter_even(data)        # [2, 4, 6, 8, 10]
average = calculate_average(data) # 5.5

Example 4: Validation Functions
def is_valid_email(email):
    return "@" in email and "." in email

def is_strong_password(password):
    return len(password) >= 8 and any(c.isupper() for c in password)

def validate_age(age):
    return 0 <= age <= 150

# Using validation functions
email = "user@example.com"
```

```
password = "MyPass123"
age = 25

print(is_valid_email(email))      # True
print(is_strong_password(password)) # True
print(validate_age(age))          # True
```

FULL BUILT-IN FUNCTIONS ARE THERE IN CHAPTER-15
========== Write your notes below==========

CHAPTER-9

Python Strings - Complete Documentation

Chapter 3: Strings - Working with Text Data

What are Strings?

Strings are one of the most fundamental and frequently used data types in Python. Think of a string as a sequence of characters - letters, numbers, symbols, and spaces - all strung together like beads on a necklace. Whether you're displaying messages to users, processing text files, or building web applications, strings are everywhere in programming.

A string in Python is a sequence of Unicode characters enclosed in quotes. This means you can work with text in any language - English, Spanish, Chinese, Arabic, or even emojis!

Key Characteristics of Strings

1. Immutability

One of the most important concepts to understand about strings is that they are immutable. This means once you create a string, you cannot change its individual characters. Any operation that appears to modify a string actually creates a new string object.

```
text = "Hello"
print(id(text))  # Shows memory address, e.g., 140234567890

text = text + " World"
print(id(text))  # Shows different memory address - new object created!
```

2. Sequence Properties

Strings are sequences, which means:

- They have a defined order (characters appear in sequence)
- You can access individual characters using indexing
- You can extract portions using slicing
- You can iterate through them character by character

3. Unicode Support

Python strings support Unicode, allowing you to work with international characters and symbols:

```
multilingual = "Hello, 你好, مرحبا, Bonjour"
emoji_text = "Python is fun! 🐍✨"
print(multilingual)
print(emoji_text)
```

Creating Strings

Different Quote Styles

Python provides three ways to create strings, each with its own use cases:

```
# Single quotes
single_quote_string = 'Hello World'

# Double quotes
double_quote_string = "Hello World"
```

```
15.
16. # Triple quotes (for multiline strings)
17. triple_quote_string = """This is a
18. multiline string that
19. spans several lines"""
20.
21. # Triple single quotes also work
22. another_multiline = '''Another way to
23. create multiline
24. strings'''
```

When to use which quotes:

- Single quotes: Most common for simple strings
- Double quotes: When your string contains single quotes (apostrophes)
- Triple quotes: For multiline strings, docstrings, or when you need both single and double quotes inside

```
25. # Using double quotes to include apostrophes
26. message = "It's a beautiful day!"
27.
28. # Using single quotes to include double quotes
29. quote = 'She said "Hello" to me'
30.
31. # Using triple quotes for complex strings
32. complex_string = """He said, "It's going to be a great day!"
33. This string contains both types of quotes."""
```

Escape Sequences

Sometimes you need to include special characters in your strings. Escape sequences use a backslash () followed by a character to represent special characters:

```
34. # Common escape sequences
35. newline_example = "First line\nSecond line"
36. tab_example = "Column1\tColumn2\tColumn3"
37. quote_example = "She said \"Hello\" to me"
38. apostrophe_example = 'It\'s working!'
39. backslash_example = "This is a backslash: \\"
40.
41. print(newline_example)
42. print(tab_example)
43. print(quote_example)
```

Complete list of escape sequences:

- \n - Newline
- \t - Tab
- \\ - Backslash
- \' - Single quote
- \" - Double quote
- \r - Carriage return
- \b - Backspace
- \f - Form feed
- \v - Vertical tab
- \0 - Null character
- \ooo - Character with octal value ooo
- \xhh - Character with hex value hh

Raw Strings

For strings that contain many backslashes (like file paths or regular expressions), you can use raw strings by prefixing with r:

```
# Regular string (backslashes need escaping)
file_path = "C:\\Users\\John\\Documents\\file.txt"

# Raw string (backslashes treated literally)
raw_path = r"C:\Users\John\Documents\file.txt"

print(file_path)
print(raw_path)  # Both print the same result
```

String Indexing and Slicing

Understanding Indexing

Python uses zero-based indexing, meaning the first character is at position 0. You can also use negative indexing to count from the end:

```
text = "Python"
#      P y t h o n
#      0 1 2 3 4 5  (positive indexing)
#     -6-5-4-3-2-1  (negative indexing)

print(text[0])   # 'P' (first character)
print(text[1])   # 'y' (second character)
print(text[-1])  # 'n' (last character)
print(text[-2])  # 'o' (second to last)
```

String Slicing

Slicing allows you to extract portions of a string using the syntax string[start:end:step]:

```
text = "Python Programming"
#      0123456789012345678  (index positions)

# Basic slicing
print(text[0:6])    # "Python" (characters 0 to 5, end excluded)
print(text[7:18])   # "Programming" (characters 7 to 17)
print(text[:6])     # "Python" (from start to position 6)
print(text[7:])     # "Programming" (from position 7 to end)

# Using step
print(text[::2])    # "Pto rgamn" (every 2nd character)
print(text[1::2])   # "yhnPoamig" (every 2nd character starting from index 1)

# Negative slicing
print(text[::-1])   # "gnimmargorP nohtyP" (reverse the string)
print(text[-11:])   # "Programming" (last 11 characters)
```

Important slicing rules:

- start is included, end is excluded
- Missing start defaults to beginning of string
- Missing end defaults to end of string
- Missing step defaults to 1
- Negative step reverses direction

String Operators

Concatenation Operator (+)

The plus operator joins strings together:

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)  # "John Doe"

# Multiple concatenations
greeting = "Hello" + " " + "beautiful" + " " + "world"
print(greeting)  # "Hello beautiful world"
```

Repetition Operator (*)

The multiplication operator repeats strings:

```
laugh = "Ha" * 5
print(laugh)  # "HaHaHaHaHa"

separator = "-" * 30
print(separator)  # "------------------------------"

# Useful for formatting
print("Title".center(20, "*"))  # "********Title********"
```

Membership Operators (in, not in)

Check if a substring exists within a string:

```
sentence = "Python is amazing"

print("Python" in sentence)     # True
print("Java" in sentence)       # False
print("Python" not in sentence) # False
print("Java" not in sentence)   # True

    # Case-sensitive checking
print("python" in sentence)     # False (different case)
    print("python" in sentence.lower())  # True (after converting to lowercase)
```

Comparison Operators

Strings can be compared using comparison operators. Python compares strings lexicographically (dictionary order):

```
    # Equality
    print("apple" == "apple")   # True
    print("apple" == "Apple")   # False (case-sensitive)

# Lexicographic comparison
    print("apple" < "banana")   # True
    print("apple" > "banana")   # False
print("apple" < "Apple")    # False (uppercase letters come first in ASCII)

# Length doesn't matter for comparison
print("a" > "zzz")        # False
print("apple" < "application")  # True
```

Complete List of String Methods

Case Conversion Methods

```
text = "Hello World Python"

# Basic case conversion
print(text.upper())       # "HELLO WORLD PYTHON"
print(text.lower())       # "hello world python"
print(text.title())       # "Hello World Python"
print(text.capitalize())  # "Hello world python"
print(text.swapcase())    # "hELLO wORLD pYTHON"

# Case checking methods
print(text.isupper())     # False
print(text.islower())     # False
print(text.istitle())     # True
print("HELLO".isupper())  # True
print("hello".islower())  # True
```

Method Details:

- upper(): Converts all characters to uppercase
- lower(): Converts all characters to lowercase
- title(): Capitalizes the first letter of each word
- capitalize(): Capitalizes only the first letter of the string
- swapcase(): Swaps the case of all characters
- isupper(): Returns True if all characters are uppercase
- islower(): Returns True if all characters are lowercase
- istitle(): Returns True if string is in title case

Whitespace Handling Methods

```
messy_text = "   Hello World   "

# Removing whitespace
print(f"'{messy_text.strip()}'")    # "'Hello World'"
print(f"'{messy_text.lstrip()}'")   # "'Hello World   '"
print(f"'{messy_text.rstrip()}'")   # "'   Hello World'"

# Custom character removal
custom_text = "***Hello World***"
```

```
print(custom_text.strip("*"))       # "Hello World"

# Checking for whitespace
print("   ".isspace())            # True
print("Hello World".isspace())     # False
```

Method Details:

- strip(): Removes whitespace from both ends
- lstrip(): Removes whitespace from the left (beginning)
- rstrip(): Removes whitespace from the right (end)
- strip(chars): Removes specified characters from both ends
- isspace(): Returns True if all characters are whitespace

Search and Find Methods

```
sentence = "Python is powerful and Python is easy to learn"

    # Finding positions
print(sentence.find("Python"))         # 0 (first occurrence)
print(sentence.find("Python", 1))      # 23 (search starting from index 1)
print(sentence.find("Java"))           # -1 (not found)

# Index method (raises exception if not found)
    print(sentence.index("Python"))        # 0
# print(sentence.index("Java"))         # Would raise ValueError

# Reverse finding
    print(sentence.rfind("Python"))        # 23 (last occurrence)
print(sentence.rindex("Python"))       # 23 (last occurrence, raises error if not found)

    # Counting occurrences
    print(sentence.count("Python"))        # 2
print(sentence.count("is"))           # 2
    print(sentence.count("the"))           # 0
```

Method Details:

- find(sub[, start[, end]]): Returns index of first occurrence, -1 if not found
- rfind(sub[, start[, end]]): Returns index of last occurrence, -1 if not found
- index(sub[, start[, end]]): Like find() but raises ValueError if not found
- rindex(sub[, start[, end]]): Like rfind() but raises ValueError if not found
- count(sub[, start[, end]]): Returns number of occurrences

Prefix and Suffix Methods

```
filename = "document.pdf"
url = "https://www.example.com"

# Checking prefixes and suffixes
print(filename.startswith("doc"))       # True
print(filename.endswith(".pdf"))        # True
print(url.startswith("https"))          # True
print(url.endswith(".com"))             # True

# Multiple options
image_file = "photo.jpg"
print(image_file.endswith((".jpg", ".png", ".gif")))  # True

# With start and end positions
text = "The Python programming language"
print(text.startswith("Python", 4))      # True (starting from index 4)
```

Method Details:

- startswith(prefix[, start[, end]]): Returns True if string starts with prefix
- endswith(suffix[, start[, end]]): Returns True if string ends with suffix
- Both methods accept tuples for checking multiple options

Content Validation Methods

```
# Testing different string types
alpha_text = "Hello"
digit_text = "12345"
alnum_text = "Hello123"
space_text = "   "
mixed_text = "Hello World!"

print("'Hello'.isalpha():", alpha_text.isalpha())       # True
print("'12345'.isdigit():", digit_text.isdigit())       # True
print("'Hello123'.isalnum():", alnum_text.isalnum())    # True
print("'   '.isspace():", space_text.isspace())        # True
print("'Hello World!'.isalpha():", mixed_text.isalpha())  # False

# Additional validation methods
print("'123'.isnumeric():", "123".isnumeric())         # True
print("'123'.isdecimal():", "123".isdecimal())         # True
print("'hello_world'.isidentifier():", "hello_world".isidentifier())  # True
```

```
print("'123abc'.isidentifier():", "123abc".isidentifier())  # False (can't start with digit)
print("'Hello World'.isprintable():", "Hello World".isprintable())  # True
```

Method Details:

- isalpha(): True if all characters are alphabetic (letters)
- isdigit(): True if all characters are digits (0-9)
- isnumeric(): True if all characters are numeric (includes digits and numeric characters)
- isdecimal(): True if all characters are decimal characters (0-9)
- isalnum(): True if all characters are alphanumeric (letters or digits)
- isspace(): True if all characters are whitespace
- isprintable(): True if all characters are printable
- isidentifier(): True if string is a valid Python identifier

String Modification Methods

```
# Replace method
original = "I love Java programming"
modified = original.replace("Java", "Python")
print(modified)  # "I love Python programming"

# Replace with count limit
text = "apple apple apple"
result = text.replace("apple", "orange", 2)  # Replace only first 2 occurrences
print(result)  # "orange orange apple"

# Split methods
data = "apple,banana,orange,grape"
fruits = data.split(",")
print(fruits)  # ['apple', 'banana', 'orange', 'grape']

# Split with limit
text = "one-two-three-four"
parts = text.split("-", 2)  # Split only on first 2 separators
print(parts)  # ['one', 'two', 'three-four']

# Right split
text = "www.example.com"
parts = text.rsplit(".", 1)  # Split from right, only once
print(parts)  # ['www.example', 'com']

# Split lines
```

```
multiline = "Line 1\nLine 2\nLine 3"
lines = multiline.splitlines()
print(lines)  # ['Line 1', 'Line 2', 'Line 3']

# Join method
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence)  # "Python is awesome"

# Join with different separators
numbers = ["1", "2", "3", "4"]
csv_format = ",".join(numbers)  # "1,2,3,4"
pipe_format = " | ".join(numbers)  # "1 | 2 | 3 | 4"
```

Method Details:

- replace(old, new[, count]): Replace occurrences of old with new
- split([sep[, maxsplit]]): Split string into list using separator
- rsplit([sep[, maxsplit]]): Split from right side
- splitlines([keepends]): Split on line breaks
- join(iterable): Join elements of iterable with string as separator

String Alignment and Formatting Methods

```
text = "Python"

# Alignment methods
print(text.center(20))        # "       Python       "
print(text.center(20, "*"))    # "*******Python*******"
print(text.ljust(20))         # "Python              "
print(text.ljust(20, "-"))     # "Python--------------"
print(text.rjust(20))         # "              Python"
print(text.rjust(20, "="))     # "==============Python"

# Zero padding (useful for numbers)
number = "42"
padded = number.zfill(5)
print(padded)  # "00042"

# Expanding tabs
tabbed_text = "Name\tAge\tCity"
expanded = tabbed_text.expandtabs(10)
print(expanded)  # "Name      Age       City"
```

Method Details:

- center(width[, fillchar]): Center string in field of given width
- ljust(width[, fillchar]): Left-justify string in field of given width
- rjust(width[, fillchar]): Right-justify string in field of given width
- zfill(width): Pad numeric string with zeros on the left
- expandtabs([tabsize]): Replace tabs with spaces

Advanced String Methods

```
# Partition methods
text = "user@domain.com"
parts = text.partition("@")
print(parts)  # ('user', '@', 'domain.com')

# Right partition
url = "https://www.example.com/page.html"
parts = url.rpartition("/")
print(parts)  # ('https://www.example.com', '/', 'page.html')

# Translation methods
text = "Hello World"
# Create translation table
trans_table = str.maketrans("aeiou", "12345")
translated = text.translate(trans_table)
print(translated)  # "H2ll4 W4rld"

# Remove characters
text = "Hello123World456"
# Remove all digits
no_digits = text.translate(str.maketrans("", "", "0123456789"))
print(no_digits)  # "HelloWorld"

# Encoding and decoding
text = "Hello, 世界!"
encoded = text.encode("utf-8")
print(encoded)  # b'Hello, \xe4\xb8\x96\xe7\x95\x8c!'
decoded = encoded.decode("utf-8")
print(decoded)  # "Hello, 世界!"
```

Method Details:

- partition(sep): Split into 3-tuple: (before_sep, sep, after_sep)
- rpartition(sep): Like partition but splits on last occurrence
- maketrans(x[, y[, z]]): Create translation table for translate()

- translate(table): Apply translation table to string
- encode([encoding[, errors]]): Encode string to bytes
- decode([encoding[, errors]]): Decode bytes to string (bytes method)

String Formatting

Old-Style Formatting (% operator)

```
name = "Alice"
age = 25
score = 87.5

# Basic formatting
old_style = "Hello %s, you are %d years old" % (name, age)
print(old_style)

# With precision
formatted_score = "Your score is %.1f%%" % score
print(formatted_score)  # "Your score is 87.5%"

# Multiple values
info = "Name: %s, Age: %d, Score: %.2f" % (name, age, score)
print(info)
```

Format specifiers:

- %s - String
- %d - Integer
- %f - Float
- %x - Hexadecimal
- %o - Octal
- %% - Literal % character

.format() Method

```
name = "Bob"
age = 30
balance = 1234.56

# Positional arguments
formatted = "Hello {}, you are {} years old".format(name, age)
print(formatted)

# Named arguments
```

```
named = "Hello {name}, your balance is ${balance:.2f}".format(name=name, balance=balance)
print(named)

# Index-based
indexed = "Hello {0}, you are {1} years old. Yes, {0}!".format(name, age)
print(indexed)

# Number formatting
print("Balance: ${:.2f}".format(balance))     # "Balance: $1234.56"
print("Balance: ${:,.2f}".format(balance))    # "Balance: $1,234.56"
print("Percentage: {:.1%}".format(0.875))     # "Percentage: 87.5%"
```

f-strings (Formatted String Literals) - Recommended

```
name = "Charlie"
age = 35
items = ["apple", "banana", "orange"]

# Basic f-string
greeting = f"Hello {name}, you are {age} years old"
print(greeting)

# Expressions inside f-strings
future_age = f"Next year you'll be {age + 1}"
item_count = f"You have {len(items)} items"
print(future_age)
print(item_count)

# Formatting numbers
price = 123.456
print(f"Price: ${price:.2f}")         # "Price: $123.46"
print(f"Price: ${price:,.2f}")        # "Price: $123.46"
print(f"Large number: {1000000:,}")   # "Large number: 1,000,000"

# Alignment in f-strings
print(f"{'Left':<10}|{'Center':^10}|{'Right':>10}")
# "Left      |  Center |     Right"

# Debug feature (Python 3.8+)
x = 10
y = 20
```

```
print(f"{x=}, {y=}, {x+y=}")  # "x=10, y=20, x+y=30"
```

String Comparison and Sorting

Basic Comparison

```
# Case-sensitive comparison
print("apple" == "Apple")      # False
print("apple".lower() == "Apple".lower())  # True

# Lexicographic ordering
fruits = ["banana", "apple", "Cherry", "date"]
print(sorted(fruits))          # ['Cherry', 'apple', 'banana', 'date']
print(sorted(fruits, key=str.lower))  # ['apple', 'banana', 'Cherry', 'date']

# Custom comparison
def compare_length(s1, s2):
    return len(s1) - len(s2)

# Sort by length
sorted_by_length = sorted(fruits, key=len)
print(sorted_by_length)  # ['date', 'apple', 'banana', 'Cherry']
```

Advanced Comparison

```
import locale

# Locale-aware comparison
words = ["café", "cafe", "naïve", "naive"]
print(sorted(words))  # Basic ASCII sorting

# Set locale for proper sorting (if available)
try:
    locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
    print(sorted(words, key=locale.strxfrm))
except:
    print("Locale not available")
```

Working with Unicode

Understanding Unicode

```
# Unicode characters
unicode_text = "Hello, 世界! 🌍"
print(unicode_text)
print(len(unicode_text))  # Length in characters, not bytes

# Unicode code points
for char in "Hello":
    print(f"'{char}': {ord(char)}")  # Get Unicode code point

# Create character from code point
print(chr(65))   # 'A'
print(chr(8364)) # '€'
print(chr(128013)) # '🌍'

# Unicode escape sequences
unicode_escape = "\u4e16\u754c"  # 世界
print(unicode_escape)
```

Encoding and Decoding

```
text = "Hello, 世界!"

# Different encodings
utf8_bytes = text.encode('utf-8')
utf16_bytes = text.encode('utf-16')
ascii_bytes = text.encode('ascii', errors='ignore')  # Ignore non-ASCII

print(f"UTF-8: {utf8_bytes}")
print(f"UTF-16: {utf16_bytes}")
print(f"ASCII (ignored): {ascii_bytes}")

# Decoding
decoded_utf8 = utf8_bytes.decode('utf-8')
print(f"Decoded: {decoded_utf8}")

# Error handling
try:
    text.encode('ascii')  # Will raise UnicodeEncodeError
```

```
except UnicodeEncodeError as e:
    print(f"Encoding error: {e}")
```

Regular Expressions with Strings

```
import re

text = "Contact us at support@company.com or call (555) 123-4567"

# Find email addresses
    emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(f"Emails found: {emails}")

    # Find phone numbers
    phones = re.findall(r'\(\d{3}\)\s\d{3}-\d{4}', text)
    print(f"Phones found: {phones}")

    # Replace patterns
    formatted = re.sub(r'\b(\w+)@(\w+)\.(\w+)\b', r'\1 AT \2 DOT \3', text)
    print(f"Formatted: {formatted}")

    # Split on multiple delimiters
    data = "apple,banana;orange:grape"
items = re.split(r'[,;:]', data)
    print(f"Items: {items}")
```

String Performance Considerations

Efficient String Building

```
import time

# Inefficient: String concatenation in loop
def build_string_slow(n):
    result = ""
    for i in range(n):
        result += str(i) + ", "
    return result[:-2]  # Remove trailing comma

# Efficient: Using join
```

```
    def build_string_fast(n):
        items = [str(i) for i in range(n)]
        return ", ".join(items)

# Timing comparison
    n = 1000
    start = time.time()
    slow_result = build_string_slow(n)
slow_time = time.time() - start

    start = time.time()
    fast_result = build_string_fast(n)
    fast_time = time.time() - start

print(f"Slow method: {slow_time:.4f} seconds")
    print(f"Fast method: {fast_time:.4f} seconds")
    print(f"Speed improvement: {slow_time/fast_time:.1f}x")
```

Memory Usage

```
    import sys

    # Compare memory usage
    short_string = "Hello"
    long_string = "Hello" * 1000

    print(f"Short string memory: {sys.getsizeof(short_string)} bytes")
    print(f"Long string memory: {sys.getsizeof(long_string)} bytes")

    # String interning
    a = "hello"
b = "hello"
    print(f"Same object: {a is b}")  # May be True due to interning

    # Force new objects
    a = "hello" + ""
    b = "hello" + ""
print(f"Same object: {a is b}")  # Likely False
```

Practical Examples

Text Processing Example

```
    def analyze_text(text):
        """Comprehensive text analysis"""
        # Clean the text
        clean_text = text.strip().lower()

        # Basic statistics
        char_count = len(text)
        char_count_no_spaces = len(text.replace(" ", ""))
        word_count = len(text.split())
        line_count = len(text.splitlines())

        # Character frequency
        char_freq = {}
        for char in clean_text:
            if char.isalpha():
                char_freq[char] = char_freq.get(char, 0) + 1

        # Most common character
        most_common = max(char_freq.items(), key=lambda x: x[1]) if char_freq else
    None

        # Word frequency
        words = clean_text.split()
        word_freq = {}
        for word in words:
            # Remove punctuation
            clean_word = ''.join(char for char in word if char.isalnum())
            if clean_word:
                word_freq[clean_word] = word_freq.get(clean_word, 0) + 1

        # Most common word
        most_common_word = max(word_freq.items(), key=lambda x: x[1]) if
    word_freq else None

        return {
     'character_count': char_count,
            'character_count_no_spaces': char_count_no_spaces,
     'word_count': word_count,
```

```
    'line_count': line_count,
    'most_common_character': most_common,
        'most_common_word': most_common_word,
    'character_frequency': char_freq,
        'word_frequency': word_freq
      }

# Example usage
sample_text = """Python is a powerful programming language.
It is easy to learn and widely used in various fields.
Python's simplicity makes it perfect for beginners."""

analysis = analyze_text(sample_text)
print("Text Analysis Results:")
print(f"Characters: {analysis['character_count']}")
print(f"Words: {analysis['word_count']}")
print(f"Lines: {analysis['line_count']}")
print(f"Most common character: {analysis['most_common_character']}")
print(f"Most common word: {analysis['most_common_word']}")
```

String Validation Example

```
def validate_input(data):
    """Validate various types of string input"""

    def validate_email(email):
        """Simple email validation"""
        if '@' not in email or '.' not in email:
            return False
        parts = email.split('@')
 if len(parts) != 2:
            return False
        local, domain = parts
        if not local or not domain:
            return False
        if not domain.count('.') >= 1:
     return False
        return True

    def validate_phone(phone):
 """Validate phone number format"""
        # Remove all non-digits
```

```
        digits_only = ''.join(char for char in phone if char.isdigit())
        return len(digits_only) == 10

    def validate_password(password):
        """Validate password strength"""
        if len(password) < 8:
            return False, "Password must be at least 8 characters"
        if not any(char.isupper() for char in password):
            return False, "Password must contain uppercase letter"
        if not any(char.islower() for char in password):
            return False, "Password must contain lowercase letter"
        if not any(char.isdigit() for char in password):
            return False, "Password must contain a number"
        if not any(char in "!@#$%^&*" for char in password):
            return False, "Password must contain special character"
        return True, "Password is strong"

    results = {}

    # Validate email
    if 'email' in data:
        results['email'] = validate_email(data['email'])

    # Validate phone
if 'phone' in data:
        results['phone'] = validate_phone(data['phone'])

    # Validate password
    if 'password' in data:
        results['password'] = validate_password(data['password'])

    return results

# Example usage
test_data = {
    'email': 'user@example.com',
    'phone': '(555) 123-4567',
    'password': 'MyStr0ng!Pass'
}

validation_results = validate_input(test_data)
for field, result in validation_results.items():
```

```
        if isinstance(result, tuple):
            status, message = result
            print(f"{field}: {'✓' if status else '✗'} {message}")
        else:
            print(f"{field}: {'✓' if result else '✗'}")
```

File Processing Example

```
    def process_log_file(log_content):
        """Process log file content and extract information"""

        lines = log_content.strip().split('\n')

        # Initialize counters
        error_count = 0
        warning_count = 0
        info_count = 0
        ip_addresses = set()
        error_messages = []

        for line in lines:
            # Skip empty lines
            if not line.strip():
                continue

    # Count log levels
            if 'ERROR' in line.upper():
                error_count += 1
                # Extract error message
                if ':' in line:
                    error_messages.append(line.split(':', 1)[1].strip())
            elif 'WARNING' in line.upper():
                warning_count += 1
            elif 'INFO' in line.upper():
                info_count += 1

            # Extract IP addresses (simple pattern)
            import re
            ip_pattern = r'\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b'
            found_ips = re.findall(ip_pattern, line)
            ip_addresses.update(found_ips)

```

```
        return {
            'total_lines': len(lines),
            'error_count': error_count,
            'warning_count': warning_count,
            'info_count': info_count,
            'unique_ip_addresses': len(ip_addresses),
            'ip_list': list(ip_addresses),
            'error_messages': error_messages[:5]  # First 5 errors
        }

    # Example log content
    sample_log = """
    2023-10-01 10:30:15 INFO Server started on port 8080
    2023-10-01 10:30:20 INFO User 192.168.1.100 connected
    2023-10-01 10:31:45 WARNING High memory usage detected
    2023-10-01 10:32:10 ERROR Database connection failed: Connection timeout
    2023-10-01 10:32:15 ERROR Failed to authenticate user from 10.0.0.5
    2023-10-01 10:33:00 INFO User 192.168.1.105 disconnected
    """

    log_analysis = process_log_file(sample_log)
    print("Log Analysis:")
    for key, value in log_analysis.items():
        print(f"{key}: {value}")
```

Complete String Methods Reference Table

Method	Description	Returns	Example
Case Conversion			
upper()	Convert to uppercase	New string	"hello".upper() → "HELLO"
lower()	Convert to lowercase	New string	"HELLO".lower() → "hello"
title()	Convert to title case	New string	"hello world".title() → "Hello World"
capitalize()	Capitalize first character	New string	"hello".capitalize() → "Hello"

swapcase()	Swap case of all characters	New string	"Hello".swapcase() → "hELLO"
Case Checking			
isupper()	Check if all uppercase	Boolean	"HELLO".isupper() → True
islower()	Check if all lowercase	Boolean	"hello".islower() → True
istitle()	Check if title case	Boolean	"Hello World".istitle() → True
Whitespace Handling			
strip([chars])	Remove from both ends	New string	" hello ".strip() → "hello"
lstrip([chars])	Remove from left end	New string	" hello".lstrip() → "hello"
rstrip([chars])	Remove from right end	New string	"hello ".rstrip() → "hello"
isspace()	Check if all whitespace	Boolean	" ".isspace() → True
Search and Find			
find(sub[, start[, end]])	Find substring position	Integer	"hello".find("ll") → 2
rfind(sub[, start[, end]])	Find from right	Integer	"hello".rfind("l") → 3
index(sub[, start[, end]])	Find (raises error if not found)	Integer	"hello".index("ll") → 2
rindex(sub[, start[, end]])	Find from right (raises error)	Integer	"hello".rindex("l") → 3
count(sub[, start[, end]])	Count occurrences	Integer	"hello".count("l") → 2
Prefix/Suffix			

startswith(prefix[, start[, end]])	Check if starts with	Boolean	"hello".startswith("he") → True
endswith(suffix[, start[, end]])	Check if ends with	Boolean	"hello".endswith("lo") → True
Content Validation			
isalpha()	Check if all alphabetic	Boolean	"hello".isalpha() → True
isdigit()	Check if all digits	Boolean	"123".isdigit() → True
isalnum()	Check if alphanumeric	Boolean	"hello123".isalnum() → True
isnumeric()	Check if numeric	Boolean	"123".isnumeric() → True
isdecimal()	Check if decimal	Boolean	"123".isdecimal() → True
isidentifier()	Check if valid identifier	Boolean	"var_name".isidentifier() → True
isprintable()	Check if printable	Boolean	"hello".isprintable() → True
String Modification			
replace(old, new[, count])	Replace occurrences	New string	"hello".replace("l", "x") → "hexxo"
split([sep[, maxsplit]])	Split into list	List	"a,b,c".split(",") → ['a', 'b', 'c']
rsplit([sep[, maxsplit]])	Split from right	List	"a.b.c".rsplit(".", 1) → ['a.b', 'c']
splitlines([keepends])	Split on line breaks	List	"a\nb".splitlines() → ['a', 'b']
join(iterable)	Join with separator	New string	",".join(['a', 'b']) → "a,b"
partition(sep)	Split into 3 parts	Tuple	"a@b.com".partition("@") → ('a', '@', 'b.com')

rpartition(sep)	Split from right into 3 parts	Tuple	"a.b.com".rpartition(".") → ('a.b', '.', 'com')
Alignment and Formatting			
center(width[, fillchar])	Center in field	New string	"hi".center(10) → " hi "
ljust(width[, fillchar])	Left justify	New string	"hi".ljust(10) → "hi "
rjust(width[, fillchar])	Right justify	New string	"hi".rjust(10) → " hi"
zfill(width)	Pad with zeros	New string	"42".zfill(5) → "00042"
expandtabs([tabsize])	Expand tabs to spaces	New string	"a\tb".expandtabs(4) → "a b"
Advanced Operations			
translate(table)	Apply translation	New string	Complex - see examples above
maketrans(x[, y[, z]])	Create translation table	Translation table	Static method
encode([encoding[, errors]])	Encode to bytes	Bytes object	"hello".encode("utf-8")
format(*args, **kwargs)	Format string	New string	"Hello {}".format("World")

String Operators Reference

Operator	Name	Description	Example	Result
+	Concatenation	Join strings	"Hello" + " World"	"Hello World"
*	Repetition	Repeat string	"Hi" * 3	"HiHiHi"
in	Membership	Check if substring exists	"lo" in "Hello"	True
not in	Non-membership	Check if substring doesn't exist	"Hi" not in "Hello"	True

==	Equality	Check if strings are equal	"Hello" == "Hello"	True
!=	Inequality	Check if strings are not equal	"Hello" != "Hi"	True
<	Less than	Lexicographic comparison	"apple" < "banana"	True
<=	Less than or equal	Lexicographic comparison	"apple" <= "apple"	True
>	Greater than	Lexicographic comparison	"banana" > "apple"	True
>=	Greater than or equal	Lexicographic comparison	"banana" >= "banana"	True
[]	Indexing	Access character at index	"Hello"[1]	"e"
[:]	Slicing	Extract substring	"Hello"[1:4]	"ell"

Common String Patterns and Idioms

Checking for Empty Strings

```
# Different ways to check for empty strings
text = ""

# Most Pythonic way
if not text:
    print("String is empty")

# Explicit comparison (less preferred)
if text == "":
    print("String is empty")

# Check for whitespace-only strings
if not text.strip():
    print("String is empty or whitespace only")
```

Safe String Operations

```
# Safe string access
```

```
def safe_get_char(text, index, default=""):
    """Safely get character at index"""
    try:
        return text[index]
    except IndexError:
        return default

# Safe string slicing
def safe_slice(text, start, end=None):
    """Safely slice string without IndexError"""
    if end is None:
        end = len(text)
    start = max(0, min(start, len(text)))
    end = max(0, min(end, len(text)))
    return text[start:end]

# Examples
text = "Hello"
print(safe_get_char(text, 10, "?"))  # "?" instead of IndexError
print(safe_slice(text, 2, 100))      # "llo" instead of going out of bounds
```

String Builder Pattern

```
# For building large strings efficiently
def build_html_table(data):
    """Build HTML table from data efficiently"""
    parts = ["<table>"]

    for row in data:
        parts.append("  <tr>")
        for cell in row:
            parts.append(f"    <td>{cell}</td>")
        parts.append("  </tr>")

    parts.append("</table>")
    return "\n".join(parts)

# Example usage
table_data = [
    ["Name", "Age", "City"],
["Alice", "25", "New York"],
["Bob", "30", "London"]
```

```
]

html_table = build_html_table(table_data)
print(html_table)
```

Important Tips and Best Practices

1. String Immutability

Remember that strings are immutable. Each operation creates a new string:

```
# This creates multiple string objects
result = ""
for i in range(1000):
    result += str(i)  # Creates new string each time

# Better approach
    result = "".join(str(i) for i in range(1000))
```

2. Use f-strings for Formatting

f-strings are the most readable and efficient for string formatting:

```
    name = "Alice"
    age = 25

    # Good
message = f"Hello {name}, you are {age} years old"

    # Also acceptable
    message = "Hello {}, you are {} years old".format(name, age)

    # Avoid (less readable)
    message = "Hello %s, you are %d years old" % (name, age)
```

3. String Comparison Best Practices

```
    # Case-insensitive comparison
    def compare_ignore_case(s1, s2):
        return s1.lower() == s2.lower()

    # Strip whitespace before comparison
```

```
    def compare_normalized(s1, s2):
        return s1.strip().lower() == s2.strip().lower()

    # Check for None values
    def safe_string_compare(s1, s2):
        if s1 is None or s2 is None:
            return s1 is s2  # Both None = True, one None = False
        return s1 == s2
```

4. Working with File Paths

```
import os

    # Use raw strings for Windows paths
    windows_path = r"C:\Users\John\Documents\file.txt"

    # Better: use os.path.join for cross-platform compatibility
    file_path = os.path.join("Users", "John", "Documents", "file.txt")

    # Or use pathlib (Python 3.4+)
    from pathlib import Path
path = Path("Users") / "John" / "Documents" / "file.txt"
```

5. String Validation Patterns

```
    def validate_string_input(value, min_length=1, max_length=None,
                              allowed_chars=None, required_chars=None):
        """Comprehensive string validation"""

        # Check type
        if not isinstance(value, str):
            return False, "Must be a string"

        # Check length
        if len(value) < min_length:
            return False, f"Must be at least {min_length} characters"

        if max_length and len(value) > max_length:
       return False, f"Must be no more than {max_length} characters"

        # Check allowed characters
```

```
    if allowed_chars:
        for char in value:
            if char not in allowed_chars:
                return False, f"Contains invalid character: {char}"

    # Check required characters
    if required_chars:
        for char in required_chars:
            if char not in value:
                return False, f"Must contain character: {char}"

    return True, "Valid"

# Example usage
is_valid, message = validate_string_input(
    "Hello123",
    min_length=5,
    max_length=20,
    allowed_chars="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
)
print(f"Validation: {message}")
```

Memory Tips for Students

Remember SIMUN

- Strings are Sequences
- Immutable (cannot change)
- Methods return new strings
- Unicode support
- Null-terminated in memory (but you don't see this)

Index and Slice Rules

- Zero-based indexing: First character is at index 0
- Negative indexing: Last character is at index -1
- Slice rule: [start:end] includes start, excludes end
- Remember: "Start In, End Out"

Method Categories

- is methods: Return Boolean (isalpha, isdigit, etc.)
- Case methods: Change case (upper, lower, title, etc.)
- Find methods: Locate substrings (find, index, count, etc.)
- Modify methods: Change content (replace, split, join, etc.)
- Format methods: Arrange text (center, ljust, rjust, etc.)

Common Mistakes to Avoid

795. Forgetting immutability:

```
# Wrong - this doesn't change the original string
text = "hello"
text.upper()  # Returns "HELLO" but doesn't modify text
print(text)   # Still prints "hello"

# Correct
text = text.upper()  # Assign the result back
```

1. Confusing find() and index():

```
# find() returns -1 if not found
pos = "hello".find("x")  # Returns -1

# index() raises exception if not found
pos = "hello".index("x")  # Raises ValueError
```

2. String concatenation in loops:

```
# Inefficient
result = ""
for item in items:
    result += str(item)

# Efficient
result = "".join(str(item) for item in items)
```

3. Not handling case sensitivity:

```
# Case-sensitive comparison
if user_input == "yes":  # Won't match "Yes" or "YES"
```

```
817. # Case-insensitive comparison
818.     if user_input.lower() == "yes":  # Matches any case
4.
```

Chapter Summary

Strings are fundamental to Python programming and are used in almost every program you'll write. Here are the key points to remember:

1. Strings are immutable sequences of Unicode characters
2. Use appropriate quotes based on content (single, double, or triple)
3. Indexing starts at 0 and supports negative indices
4. Slicing uses [start:end:step] syntax with end excluded
5. Rich set of methods available for manipulation and validation
6. f-strings are preferred for string formatting
7. Join method is efficient for building strings from sequences
8. Always consider Unicode when working with international text
9. Use appropriate comparison methods for your use case
10. Remember performance implications of string operations

Understanding strings thoroughly will make you more effective at text processing, user input validation, file handling, and many other common programming tasks. Practice with the examples provided, and don't hesitate to experiment with different string methods to see how they work!

CHAPTER-10

Input and Output in Python

Definition

Input and Output (I/O) in Python refers to the mechanism of receiving data from users or external sources (input) and displaying or sending data to users or external destinations (output). Python provides built-in functions and methods to handle these operations efficiently.

Key Points to Remember

1. input() Function - Getting User Input

- The input() function always returns data as a string type
- It pauses program execution until the user enters data and presses Enter
- Syntax: variable = input("prompt message")
- Memory Tip: "input() = string output" - always converts to string

```
name = input("Enter your name: ")
age = input("Enter your age: ")  # This is still a string!
age = int(input("Enter your age: "))  # Convert to integer
```

2. print() Function - Displaying Output

- Default behavior: adds a newline (\n) at the end
- Can print multiple values separated by spaces by default
- Parameters to remember: sep, end, file, flush
- Memory Tip: "print() parameters - SELF" (Sep, End, fLush, File)

```
print("Hello", "World")  # Output: Hello World
```

```
print("Hello", "World", sep="-")  # Output: Hello-World
print("Hello", end=" ")
print("World")  # Output: Hello World (on same line)
```

3. Type Conversion for Input

- input() returns string, so convert when needed
- Common conversions: int(), float(), bool(), eval()
- Memory Tip: "IFE" - Int, Float, Eval for numeric inputs

```
# Getting different data types
number = int(input("Enter a number: "))
price = float(input("Enter price: "))
multiple_nums = eval(input("Enter numbers: "))  # Can input: 1,2,3
```

4. Formatted Output Methods

- Old style: % formatting (less preferred)
- New style: .format() method
- Latest: f-strings (Python 3.6+) - most preferred
- Memory Tip: "% < .format() < f-strings" (evolution order)

```
name = "Alice"
age = 25

# f-strings (recommended)
print(f"Name: {name}, Age: {age}")

# .format() method
print("Name: {}, Age: {}".format(name, age))

# % formatting
print("Name: %s, Age: %d" % (name, age))
```

5. File Input/Output Basics

- Use open() function with modes: 'r' (read), 'w' (write), 'a' (append)
- Always close files with close() or use with statement
- Memory Tip: "RWA" - Read, Write, Append modes

```
# Writing to file
with open("data.txt", "w") as file:
    file.write("Hello World")
```

```
# Reading from file
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

6. Command Line Arguments

- Import sys module to access sys.argv
- sys.argv[0] is always the script name
- Remaining elements are the arguments passed
- Memory Tip: "argv[0] = script name, rest = arguments"

```
import sys

print("Script name:", sys.argv[0])
print("Arguments:", sys.argv[1:])
# Run: python script.py arg1 arg2
```

Common Exam Patterns

Input Validation

```
while True:
    try:
        age = int(input("Enter age: "))
        if age > 0:
            break
        else:
            print("Age must be positive")
    except ValueError:
        print("Please enter a valid number")
```

Multiple Inputs in One Line

```
# Method 1: Using split()
a, b, c = input("Enter three numbers: ").split()
a, b, c = int(a), int(b), int(c)

# Method 2: Using map()
a, b, c = map(int, input("Enter three numbers: ").split())
```

```
# Method 3: List comprehension
numbers = [int(x) for x in input("Enter numbers: ").split()]
```

Formatted Output Examples

```
# Decimal places
pi = 3.14159
print(f"Pi rounded: {pi:.2f}")  # Output: Pi rounded: 3.14

# Padding and alignment
name = "Python"
print(f"{name:>10}")   # Right align in 10 spaces
print(f"{name:<10}")   # Left align in 10 spaces
print(f"{name:^10}")   # Center align in 10 spaces
```

Quick Reference for Exams

Input Functions:

- input() - string input
- int(input()) - integer input
- float(input()) - float input
- eval(input()) - evaluates expression

Output Functions:

- print() - basic output
- print(sep="") - custom separator
- print(end="") - custom ending
- f"{variable}" - f-string formatting

File Operations:

- open(file, mode) - open file
- file.read() - read entire file
- file.readline() - read one line
- file.write(data) - write to file

Remember: Always handle exceptions for input operations and close files properly!

Exception Handling in Python

Definition

Exception Handling in Python is a programming construct that allows developers to anticipate, catch, and manage runtime errors gracefully without crashing the program. It uses try-except blocks to handle exceptions (errors) that occur during program execution, ensuring the program continues running or terminates gracefully.

Key Points to Remember

1. Basic Try-Except Structure

- try block contains code that might raise an exception
- except block handles the exception if it occurs
- Program continues after except block (doesn't crash)
- Memory Tip: "TRY to do something, EXCEPT when it fails"

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
    print(f"Result: {result}")
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

2. Exception Hierarchy and Common Types

- All exceptions inherit from BaseException class
- Most common parent: Exception class
- Memory Tip: "VIZO-NAME" - ValueError, IndexError, ZeroDivisionError, OSError, NameError, AttributeError, ModuleNotFoundError, Exception

```
# Common Exception Types:
# ValueError - wrong value type
# IndexError - list index out of range
# ZeroDivisionError - division by zero
# KeyError - dictionary key not found
# FileNotFoundError - file doesn't exist
```

```
# TypeError - wrong data type operation
```

3. Try-Except-Else-Finally Structure

- else: executes only if NO exception occurs in try block
- finally: ALWAYS executes regardless of exceptions
- Memory Tip: "TEEF" - Try, Except, Else, Finally (execution order)

```
try:
    file = open("data.txt", "r")
    data = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully!")  # Only if no exception
finally:
    print("Cleanup code here")  # Always executes
    if 'file' in locals():
        file.close()
```

4. Multiple Exception Handling

- Can handle multiple exceptions in one except block using tuple
- Can have multiple except blocks for different exceptions
- More specific exceptions should come before general ones
- Memory Tip: "Specific to General" - like a funnel shape

```
try:
    value = int(input("Enter number: "))
    result = 10 / value
    my_list = [1, 2, 3]
    print(my_list[value])
except (ValueError, TypeError):  # Multiple in one block
    print("Input or type error!")
except ZeroDivisionError:  # Specific exception
    print("Cannot divide by zero!")
except Exception as e:  # General exception (should be last)
    print(f"An error occurred: {e}")
```

5. Raising Custom Exceptions

- Use raise keyword to manually trigger exceptions
- Can raise built-in exceptions or create custom ones

- Custom exceptions should inherit from Exception class
- Memory Tip: "RAISE the alarm when something's wrong"

```
# Raising built-in exceptions
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    if age > 150:
        raise ValueError("Age seems unrealistic!")

# Custom exception class
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Using custom exception
def validate_password(password):
    if len(password) < 8:
        raise CustomError("Password must be at least 8 characters!")
```

6. Exception Information and Debugging

- Use as keyword to capture exception object
- Exception object contains error details and traceback
- str(e) gives error message, type(e) gives exception type
- Memory Tip: "AS you catch, you can examine"

```
import traceback

try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error type: {type(e).__name__}")
    print(f"Error message: {str(e)}")
    print(f"Traceback:")
    traceback.print_exc()  # Prints detailed traceback
```

7. Best Practices for Exception Handling

- Be specific with exception types (avoid bare except:)
- Don't ignore exceptions silently
- Use finally for cleanup operations

- Memory Tip: "SLF" - Specific, Log, Finally (best practices)

```
# Good practice
try:
    with open("file.txt", "r") as f:
        data = f.read()
except FileNotFoundError:
    print("File not found, creating new one...")
    with open("file.txt", "w") as f:
        f.write("Default content")
except PermissionError:
    print("Permission denied to access file")

# Bad practice - avoid this
try:
    # some code
    pass
except:  # Too general, catches everything
    pass  # Silently ignores errors
```

Common Exam Patterns

Input Validation with Exception Handling

```
def get_valid_integer():
    while True:
        try:
            value = int(input("Enter an integer: "))
            return value
        except ValueError:
            print("Invalid input! Please enter a valid integer.")

# Usage
number = get_valid_integer()
print(f"You entered: {number}")
```

File Operations with Exception Handling

```
def safe_file_read(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
```

```
    except FileNotFoundError:
        print(f"File '{filename}' not found.")
        return None
    except PermissionError:
        print(f"Permission denied to read '{filename}'.")
        return None
    except Exception as e:
        print(f"Unexpected error: {e}")
        return None
```

Calculator with Exception Handling

```
def safe_calculator():
    try:
        a = float(input("Enter first number: "))
        operator = input("Enter operator (+, -, *, /): ")
        b = float(input("Enter second number: "))

        if operator == '+':
            result = a + b
        elif operator == '-':
            result = a - b
        elif operator == '*':
            result = a * b
        elif operator == '/':
            if b == 0:
                raise ZeroDivisionError("Division by zero is not allowed!")
            result = a / b
        else:
            raise ValueError("Invalid operator!")

        print(f"Result: {result}")

    except ValueError as e:
        print(f"Value Error: {e}")
    except ZeroDivisionError as e:
        print(f"Math Error: {e}")
    except Exception as e:
        print(f"Unexpected error: {e}")
```

Quick Reference for Exams

Basic Structure:

```
try:
   # risky code
except SpecificException:
   # handle specific error
except Exception as e:
   # handle any other error
else:
   # runs if no exception
finally:
   # always runs
```

Common Exception Types:

- ValueError - wrong value
- TypeError - wrong type
- IndexError - list index error
- KeyError - dictionary key error
- ZeroDivisionError - division by zero
- FileNotFoundError - file doesn't exist

Key Keywords:

- try - attempt risky code
- except - catch exceptions
- else - runs if no exception
- finally - always runs
- raise - manually raise exception
- as - capture exception object

Remember: Always handle exceptions specifically, never use bare except:, and use finally for cleanup operations!

CHAPTER-11

Modules and Packages in Python

Definition

Module: A single Python file (.py) containing Python code (functions, classes, variables) that can be imported and reused in other programs. Package: A collection of related modules organized in a directory with an __init__.py file, creating a hierarchical structure for better code organization.

Key Points to Remember

1. Module Creation and Import

- Any .py file is automatically a module
- Import using import, from...import, or import...as
- Memory Tip: "IFA" - Import, From, As (three ways to import)

```
# Creating a module (mymodule.py)
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159

# Using the module (main.py)
import mymodule
print(mymodule.greet("Alice"))
print(mymodule.PI)

# Alternative imports
from mymodule import greet, PI
import mymodule as mm
```

2. Built-in vs User-defined Modules

- Built-in: Pre-installed with Python (math, random, os, sys)
- User-defined: Created by programmers
- Third-party: Installed using pip (requests, numpy, pandas)
- Memory Tip: "BUT" - Built-in, User-defined, Third-party

```
# Built-in modules
import math
import random
import os

# Usage
print(math.sqrt(16))  # 4.0
print(random.randint(1, 10))  # Random number
print(os.getcwd())  # Current directory
```

3. Package Structure and init.py

- Package = directory with __init__.py file
- __init__.py can be empty or contain initialization code
- Allows hierarchical module organization
- Memory Tip: "Package needs INIT to be legit"

```
# Package structure:
mypackage/
  __init__.py
  module1.py
  module2.py
  subpackage/
    __init__.py
    module3.py

# Importing from package
from mypackage import module1
from mypackage.subpackage import module3
import mypackage.module2 as m2
```

4. Module Search Path and sys.path

- Python searches modules in specific order
- Current directory → PYTHONPATH → Standard library → Site-packages

- sys.path shows the search path list
- Memory Tip: "CPSS" - Current, PythonPath, Standard, Site-packages

```
import sys
print(sys.path)  # Shows module search directories

# Adding custom path
sys.path.append('/path/to/custom/modules')
```

5. Special Module Attributes

- __name__: Module's name (or 'main' if run directly)
- __file__: Module's file path
- dir(): Lists all attributes in module
- Memory Tip: "NFD" - Name, File, Dir (special attributes)

```
# In any module
print(__name__)  # Module name
print(__file__)  # File path
print(dir())     # All attributes

# Common pattern
if __name__ == "__main__":
    # Code runs only when file is executed directly
    print("Running as main program")
```

Common Exam Patterns

Module Creation Example

```
# calculator.py (module)
def add(a, b):
    return a + b

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

def multiply(a, b):
    return a * b

def divide(a, b):
    if b != 0:
        return a / b
```

```
    else:
        return "Cannot divide by zero"

# main.py (using the module)
import calculator

result1 = calculator.add(5, 3)
result2 = calculator.divide(10, 2)
print(f"Addition: {result1}, Division: {result2}")
```

Package Example

```
# mathops/__init__.py
from .basic import add, subtract
from .advanced import power, factorial

# mathops/basic.py
def add(a, b):
    return a + b

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

# mathops/advanced.py
def power(base, exp):
    return base ** exp

def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

# Using the package
from mathops import add, power
print(add(5, 3))      # 8
print(power(2, 3))    # 8
```

Conditional Import and Error Handling

```
try:
    import numpy as np
```

```
    print("NumPy is available")
except ImportError:
    print("NumPy is not installed")

# Conditional execution
if __name__ == "__main__":
    # Test code here
    print("Testing module functions...")
```

Quick Reference for Exams

Import Methods:

- import module - imports entire module
- from module import function - imports specific items
- import module as alias - imports with alias
- from module import * - imports everything (not recommended)

Key Concepts:

- Module = single .py file
- Package = directory with init.py
- __name__ == "__main__" - checks if run directly
- sys.path - module search path

Built-in Modules to Know:

- math - mathematical functions
- random - random number generation
- os - operating system interface
- sys - system-specific parameters
- datetime - date and time handling

Package Structure:

```
package/
    __init__.py    # Makes it a package
    module1.py     # Individual modules
    module2.py
    subpackage/    # Nested packages
        __init__.py
        module3.py
```

Remember: Modules promote code reusability, packages organize modules hierarchically, and __init__.py makes a directory a package!

CHAPTER-12

File Handling in Python - Detailed Guide

Definition

File Handling in Python refers to the process of working with files stored on disk - creating, opening, reading, writing, modifying, and closing files. Python provides built-in functions and methods to perform these operations efficiently, allowing programs to store data permanently and retrieve it when needed.

Key Points to Remember

1. File Opening Modes - The Foundation

- Python uses different modes to specify how a file should be opened
- Text modes: 'r', 'w', 'a', 'r+', 'w+', 'a+'
- Binary modes: 'rb', 'wb', 'ab', 'rb+', 'wb+', 'ab+'
- Memory Tip: "RWA" - Read, Write, Append (basic modes) + "+" for read-write

```
# Text modes (default)
file1 = open('data.txt', 'r')    # Read only (file must exist)
file2 = open('data.txt', 'w')    # Write only (creates/overwrites)
file3 = open('data.txt', 'a')    # Append only (creates if not exists)
file4 = open('data.txt', 'r+')   # Read and write (file must exist)
file5 = open('data.txt', 'w+')   # Write and read (creates/overwrites)
file6 = open('data.txt', 'a+')   # Append and read (creates if not exists)

# Binary modes (for images, videos, etc.)
file7 = open('image.jpg', 'rb')  # Read binary
file8 = open('image.jpg', 'wb')  # Write binary
```

2. File Opening and Closing - Proper Resource Management

- Always close files to free system resources
- Use close() method or with statement (recommended)

- with statement automatically closes files (context manager)
- Memory Tip: "WITH wisdom, files close automatically"

```
# Method 1: Manual closing (not recommended)
file = open('data.txt', 'r')
content = file.read()
file.close()  # Must remember to close

# Method 2: with statement (recommended)
with open('data.txt', 'r') as file:
    content = file.read()
# File automatically closed here

# Multiple files with 'with'
with open('input.txt', 'r') as infile, open('output.txt', 'w') as outfile:
    data = infile.read()
    outfile.write(data.upper())
```

3. Reading Methods - Different Ways to Extract Data

- read(): Reads entire file as single string
- readline(): Reads one line at a time
- readlines(): Reads all lines into a list
- Memory Tip: "R-R-R" - Read (all), Readline (one), Readlines (list)

```
# Reading entire file
with open('data.txt', 'r') as file:
    content = file.read()  # Returns entire file as string
    print(content)

# Reading line by line
with open('data.txt', 'r') as file:
    first_line = file.readline()  # Returns first line with \n
    second_line = file.readline()  # Returns second line with \n

# Reading all lines into list
with open('data.txt', 'r') as file:
    lines = file.readlines()  # Returns list of lines (with \n)
    for line in lines:
        print(line.strip())  # strip() removes \n

# Iterating through file (memory efficient)
with open('data.txt', 'r') as file:
    for line in file:  # File object is iterable
```

```
    print(line.strip())
```

4. Writing Methods - Different Ways to Store Data

- write(): Writes string to file
- writelines(): Writes list of strings to file
- Write mode ('w') overwrites, append mode ('a') adds to end
- Memory Tip: "Write writes strings, Writelines writes lists"

```
# Writing single string
with open('output.txt', 'w') as file:
    file.write("Hello World!\n")
    file.write("Second line\n")

# Writing multiple lines using writelines()
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open('output.txt', 'w') as file:
    file.writelines(lines)

# Appending to existing file
with open('output.txt', 'a') as file:
    file.write("This is appended\n")

# Writing with print() function
with open('output.txt', 'w') as file:
    print("Hello World!", file=file)
    print("Another line", file=file)
```

5. File Pointer and Position Control

- File pointer tracks current position in file
- tell(): Returns current position
- seek(offset, whence): Moves pointer to specific position
- Memory Tip: "TELL me where, SEEK to move" + "SWE" - Start, Where, End (whence values)

```
with open('data.txt', 'r+') as file:
    print("Initial position:", file.tell())  # Usually 0

    content = file.read(5)  # Read 5 characters
    print("After reading 5 chars:", file.tell())  # Position 5

    file.seek(0)  # Move to beginning
```

```
    print("After seek(0):", file.tell())  # Position 0

    file.seek(0, 2)  # Move to end (0 offset from end)
    print("At end:", file.tell())  # File size

    # whence values:
    # 0 - from beginning (default)
    # 1 - from current position
    # 2 - from end
```

6. File Properties and Metadata

- Check file existence, size, permissions
- Use os.path and os.stat modules
- File object attributes: name, mode, closed
- Memory Tip: "NMC" - Name, Mode, Closed (file object attributes)

```
import os
import os.path

# File object attributes
with open('data.txt', 'r') as file:
    print("File name:", file.name)
    print("File mode:", file.mode)
    print("Is closed:", file.closed)  # False while open

print("Is closed after with:", file.closed)  # True after with block

# File system operations
filename = 'data.txt'
if os.path.exists(filename):
    print("File exists")
    print("File size:", os.path.getsize(filename), "bytes")
    print("Is file:", os.path.isfile(filename))
    print("Is directory:", os.path.isdir(filename))
    print("Absolute path:", os.path.abspath(filename))
```

7. Binary File Handling

- Used for non-text files (images, videos, executables)
- Use 'b' in mode ('rb', 'wb', 'ab')
- Data handled as bytes objects

- Memory Tip: "Binary = Bytes, Text = Strings"

```
# Reading binary file
with open('image.jpg', 'rb') as file:
    binary_data = file.read()
    print(f"File size: {len(binary_data)} bytes")
    print(f"First 10 bytes: {binary_data[:10]}")

# Writing binary file (copying)
with open('image.jpg', 'rb') as source:
    with open('copy.jpg', 'wb') as destination:
        destination.write(source.read())

# Working with bytes
data = b"Hello World"  # bytes literal
with open('binary.dat', 'wb') as file:
    file.write(data)
```

8. CSV File Handling

- CSV (Comma Separated Values) for structured data
- Use built-in csv module for proper handling
- Handles quotes, escapes, different delimiters
- Memory Tip: "CSV = Comma Separated Values = Structured data"

```
import csv

# Writing CSV
data = [
    ['Name', 'Age', 'City'],
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'London'],
    ['Charlie', 35, 'Tokyo']
]

with open('people.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)

# Reading CSV
with open('people.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)  # Each row is a list
```

```
# CSV with dictionaries
with open('people.csv', 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(f"Name: {row['Name']}, Age: {row['Age']}")
```

Advanced File Handling Concepts

Exception Handling with Files

```
def safe_file_operation(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            return content
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return None
    except PermissionError:
        print(f"Error: Permission denied for '{filename}'")
        return None
    except IOError as e:
        print(f"Error: I/O operation failed - {e}")
        return None

# Usage
content = safe_file_operation('data.txt')
if content:
    print("File read successfully")
```

Working with Large Files

```
def process_large_file(filename):
    """Process large files line by line to save memory"""
    try:
        with open(filename, 'r') as file:
            line_count = 0
            for line in file:  # Memory efficient iteration
                # Process each line
                line = line.strip()
```

```
            if line:  # Skip empty lines
                line_count += 1
                # Do something with line

        print(f"Processed {line_count} lines")
    except Exception as e:
        print(f"Error processing file: {e}")

# Reading file in chunks
def read_in_chunks(filename, chunk_size=1024):
    with open(filename, 'r') as file:
        while True:
            chunk = file.read(chunk_size)
            if not chunk:
                break
            # Process chunk
            yield chunk
```

File Backup and Versioning

```
import shutil
import datetime

def backup_file(filename):
    """Create backup with timestamp"""
    if os.path.exists(filename):
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        backup_name = f"{filename}.backup_{timestamp}"
        shutil.copy2(filename, backup_name)
        print(f"Backup created: {backup_name}")
        return backup_name
    return None

def safe_write_with_backup(filename, content):
    """Write to file with automatic backup"""
    backup_name = backup_file(filename)
    try:
        with open(filename, 'w') as file:
            file.write(content)
        print("File written successfully")
    except Exception as e:
        print(f"Error writing file: {e}")
```

```
        if backup_name and os.path.exists(backup_name):
            shutil.copy2(backup_name, filename)
            print("File restored from backup")
```

Common Exam Patterns

1. File Statistics Program

```
def file_statistics(filename):
    """Calculate various file statistics"""
    try:
        with open(filename, 'r') as file:
            content = file.read()
            lines = content.split('\n')
            words = content.split()

            stats = {
                'characters': len(content),
                'words': len(words),
                'lines': len(lines),
                'paragraphs': len([p for p in content.split('\n\n') if p.strip()])
            }

            return stats
    except FileNotFoundError:
        return None

# Usage
stats = file_statistics('document.txt')
if stats:
    print(f"Characters: {stats['characters']}")
    print(f"Words: {stats['words']}")
    print(f"Lines: {stats['lines']}")
```

2. File Search and Replace

```
def search_replace_in_file(filename, search_text, replace_text):
    """Search and replace text in file"""
    try:
        # Read original content
        with open(filename, 'r') as file:
```

```
            content = file.read()

        # Check if search text exists
        if search_text in content:
            # Replace text
            new_content = content.replace(search_text, replace_text)

            # Write back to file
            with open(filename, 'w') as file:
                file.write(new_content)

            return True
        else:
            print(f"Text '{search_text}' not found in file")
            return False

    except Exception as e:
        print(f"Error: {e}")
        return False

# Usage
success = search_replace_in_file('data.txt', 'old_text', 'new_text')
```

3. Log File Analyzer

```
def analyze_log_file(filename):
    """Analyze log file for patterns"""
    error_count = 0
    warning_count = 0
    info_count = 0

    try:
        with open(filename, 'r') as file:
            for line_num, line in enumerate(file, 1):
                line = line.strip().lower()
                if 'error' in line:
                    error_count += 1
                elif 'warning' in line:
                    warning_count += 1
                elif 'info' in line:
                    info_count += 1
```

```
        print(f"Log Analysis Results:")
        print(f"Errors: {error_count}")
        print(f"Warnings: {warning_count}")
        print(f"Info: {info_count}")

    except Exception as e:
        print(f"Error analyzing log: {e}")

# Usage
analyze_log_file('application.log')
```

Quick Reference for Exams

File Opening Modes:

- 'r' - Read (default)
- 'w' - Write (overwrites)
- 'a' - Append
- 'r+' - Read + Write
- 'rb', 'wb' - Binary modes

Reading Methods:

- read() - entire file
- readline() - one line
- readlines() - all lines as list
- for line in file: - iterate

Writing Methods:

- write(string) - write string
- writelines(list) - write list of strings

File Position:

- tell() - current position
- seek(position) - move to position

Best Practices:

- Always use with statement
- Handle exceptions (FileNotFoundError, PermissionError)
- Close files properly
- Use appropriate mode for operation
- Strip newlines when reading lines

Remember: File handling is about CRUD operations - Create, Read, Update, Delete files with proper resource management!

========== Write your notes below==========

CHAPTER-13

Object-Oriented Programming in Python - Complete Master Guide

Definition

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects and classes rather than functions and logic. It models real-world entities as objects with attributes (data) and methods (behavior), promoting code reusability, modularity, and maintainability through four fundamental principles: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Core OOP Concepts - The Foundation

1. Classes and Objects - The Building Blocks

- Class: Blueprint or template that defines attributes and methods
- Object: Instance of a class with actual data and behavior
- Memory Tip: "Class is COOKIE CUTTER, Object is the COOKIE"

```
# Class Definition
class Car:
    # Class attribute (shared by all instances)
    wheels = 4

    # Constructor method (__init__)
    def __init__(self, brand, model, year):
        # Instance attributes (unique to each object)
        self.brand = brand
        self.model = model
        self.year = year
        self.is_running = False

    # Instance method
```

```
    def start_engine(self):
        self.is_running = True
        return f"{self.brand} {self.model} engine started!"

    # Instance method with parameters
    def drive(self, distance):
        if self.is_running:
            return f"Driving {distance} km in {self.brand} {self.model}"
        else:
            return "Start the engine first!"

    # String representation method
    def __str__(self):
        return f"{self.year} {self.brand} {self.model}"

# Creating objects (instantiation)
car1 = Car("Toyota", "Camry", 2023)
car2 = Car("Honda", "Civic", 2022)

# Accessing attributes
print(car1.brand)  # Toyota
print(Car.wheels)  # 4 (class attribute)

# Calling methods
print(car1.start_engine())  # Toyota Camry engine started!
print(car1.drive(50))      # Driving 50 km in Toyota Camry
print(car1)                # 2023 Toyota Camry
```

2. Encapsulation - Data Protection and Access Control

- Bundles data and methods that operate on that data within a single unit
- Controls access through public, protected, and private members
- Memory Tip: "PPP" - Public (no underscore), Protected (_), Private (__)

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        # Public attribute
        self.account_number = account_number

        # Protected attribute (convention: one underscore)
        self._account_type = "Savings"

        # Private attribute (name mangling: two underscores)
```

```
    self.__balance = initial_balance
    self.__pin = 1234

# Public method
def get_balance(self):
    return self.__balance

# Public method with validation
def deposit(self, amount):
    if amount > 0:
        self.__balance += amount
        return f"Deposited ${amount}. New balance: ${self.__balance}"
    else:
        return "Invalid deposit amount"

# Private method (internal use only)
def __validate_pin(self, pin):
    return pin == self.__pin

# Public method using private method
def withdraw(self, amount, pin):
    if not self.__validate_pin(pin):
        return "Invalid PIN"

    if amount > 0 and amount <= self.__balance:
        self.__balance -= amount
        return f"Withdrew ${amount}. Remaining balance: ${self.__balance}"
    else:
        return "Invalid withdrawal amount or insufficient funds"

# Property decorator for controlled access
@property
def balance(self):
    return self.__balance

# Setter with validation
@balance.setter
def balance(self, value):
    if value >= 0:
        self.__balance = value
    else:
        raise ValueError("Balance cannot be negative")
```

```
# Usage
account = BankAccount("ACC123", 1000)

# Public access
print(account.account_number)  # ACC123

# Protected access (possible but not recommended)
print(account._account_type)   # Savings

# Private access (will cause AttributeError)
# print(account.__balance)     # Error!
# print(account.__pin)         # Error!

# Proper access through methods
print(account.get_balance())   # 1000
print(account.deposit(500))    # Deposited $500. New balance: $1500
print(account.withdraw(200, 1234))  # Withdrew $200. Remaining balance: $1300

# Using property
print(account.balance)         # 1300 (getter)
account.balance = 1500         # setter
```

3. Inheritance - Code Reusability and Hierarchy

- Allows a class to inherit attributes and methods from another class
- Parent/Base/Super class provides common functionality
- Child/Derived/Sub class extends or modifies parent functionality
- Memory Tip: "IS-A relationship" - Child IS-A type of Parent

```
# Base class (Parent)
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.is_alive = True

    def eat(self):
        return f"{self.name} is eating"

    def sleep(self):
        return f"{self.name} is sleeping"
```

```
    def make_sound(self):
        return "Some generic animal sound"

    # Method that can be overridden
    def move(self):
        return f"{self.name} is moving"

# Single Inheritance
class Dog(Animal):
    def __init__(self, name, breed):
        # Call parent constructor
        super().__init__(name, "Canine")
        self.breed = breed
        self.tricks = []

    # Override parent method
    def make_sound(self):
        return f"{self.name} says Woof!"

    # Add new method
    def learn_trick(self, trick):
        self.tricks.append(trick)
        return f"{self.name} learned {trick}"

    # Override with extension
    def move(self):
        return f"{self.name} is running on four legs"

class Bird(Animal):
    def __init__(self, name, can_fly=True):
        super().__init__(name, "Avian")
        self.can_fly = can_fly

    def make_sound(self):
        return f"{self.name} chirps"

    def move(self):
        if self.can_fly:
            return f"{self.name} is flying"
        else:
            return f"{self.name} is walking"
```

```
# Multiple Inheritance
class Mammal:
    def __init__(self):
        self.warm_blooded = True
        self.has_fur = True

    def give_birth(self):
        return "Gives birth to live young"

class Aquatic:
    def __init__(self):
        self.can_swim = True

    def swim(self):
        return "Swimming in water"

class Dolphin(Mammal, Aquatic, Animal):
    def __init__(self, name):
        Animal.__init__(self, name, "Cetacean")
        Mammal.__init__(self)
        Aquatic.__init__(self)

    def make_sound(self):
        return f"{self.name} clicks and whistles"

    def move(self):
        return f"{self.name} is swimming gracefully"

# Usage Examples
dog = Dog("Buddy", "Golden Retriever")
bird = Bird("Tweety")
dolphin = Dolphin("Flipper")

# Inherited methods
print(dog.eat())         # Buddy is eating
print(bird.sleep())      # Tweety is sleeping

# Overridden methods
print(dog.make_sound())    # Buddy says Woof!
print(bird.make_sound())   # Tweety chirps
print(dolphin.make_sound()) # Flipper clicks and whistles
```

```
# New methods
print(dog.learn_trick("sit"))  # Buddy learned sit

# Multiple inheritance
print(dolphin.swim())         # Swimming in water
print(dolphin.give_birth())    # Gives birth to live young

# Method Resolution Order (MRO)
print(Dolphin.mro())  # Shows inheritance hierarchy
```

4. Polymorphism - One Interface, Multiple Forms

- Same interface behaves differently for different object types
- Runtime Polymorphism: Method overriding (different implementations)
- Compile-time Polymorphism: Method overloading (Python uses default parameters)
- Memory Tip: "POLY = MANY, MORPH = FORMS" - Many forms of same interface

```
# Polymorphism through inheritance
class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        pass  # Abstract method to be overridden

    def perimeter(self):
        pass  # Abstract method to be overridden

    def describe(self):
        return f"This is a {self.name}"

class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)
```

```
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

class Triangle(Shape):
    def __init__(self, a, b, c):
        super().__init__("Triangle")
        self.a = a
        self.b = b
        self.c = c

    def area(self):
        # Heron's formula
        s = (self.a + self.b + self.c) / 2
        return (s * (s - self.a) * (s - self.b) * (s - self.c)) ** 0.5

    def perimeter(self):
        return self.a + self.b + self.c

# Polymorphic function
def calculate_shape_info(shape):
    """Works with any Shape object"""
    print(f"Shape: {shape.describe()}")
    print(f"Area: {shape.area():.2f}")
    print(f"Perimeter: {shape.perimeter():.2f}")
    print("-" * 30)

# Polymorphism in action
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(3, 4, 5)
]
```

```
for shape in shapes:
    calculate_shape_info(shape)  # Same function, different behavior

# Duck Typing (Python's approach to polymorphism)
class Duck:
    def make_sound(self):
        return "Quack!"

    def swim(self):
        return "Swimming like a duck"

class Robot:
    def make_sound(self):
        return "Beep beep!"

    def swim(self):
        return "Swimming with mechanical precision"

def test_duck_like_behavior(obj):
    """If it quacks like a duck and swims like a duck..."""
    print(obj.make_sound())
    print(obj.swim())

# Duck typing in action
duck = Duck()
robot = Robot()

test_duck_like_behavior(duck)   # Works
test_duck_like_behavior(robot)  # Also works!
```

5. Abstraction - Hiding Implementation Details

- Hides complex implementation details and shows only essential features
- Uses abstract classes and interfaces
- ABC module: Abstract Base Classes in Python
- Memory Tip: "ABSTRACT = Show WHAT, hide HOW"

```
from abc import ABC, abstractmethod

# Abstract base class
class Vehicle(ABC):
    def __init__(self, brand, model):
        self.brand = brand
```

```
        self.model = model
        self.is_running = False

    # Abstract method (must be implemented by subclasses)
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

    @abstractmethod
    def accelerate(self):
        pass

    # Concrete method (can be used as-is or overridden)
    def get_info(self):
        return f"{self.brand} {self.model}"

    # Template method pattern
    def start_journey(self):
        print(f"Starting journey with {self.get_info()}")
        self.start_engine()
        self.accelerate()
        print("Journey started!")

class Car(Vehicle):
    def __init__(self, brand, model, fuel_type):
        super().__init__(brand, model)
        self.fuel_type = fuel_type

    def start_engine(self):
        self.is_running = True
        return f"Car engine started with {self.fuel_type}"

    def stop_engine(self):
        self.is_running = False
        return "Car engine stopped"

    def accelerate(self):
        return "Car is accelerating smoothly"
```

```
class Motorcycle(Vehicle):
    def __init__(self, brand, model, engine_cc):
        super().__init__(brand, model)
        self.engine_cc = engine_cc

    def start_engine(self):
        self.is_running = True
        return f"Motorcycle engine ({self.engine_cc}cc) roars to life!"

    def stop_engine(self):
        self.is_running = False
        return "Motorcycle engine stopped"

    def accelerate(self):
        return "Motorcycle is accelerating rapidly"

# Usage
# vehicle = Vehicle("Generic", "Vehicle")  # Error! Cannot instantiate abstract class

car = Car("Toyota", "Prius", "Hybrid")
bike = Motorcycle("Yamaha", "R1", 1000)

car.start_journey()
print(car.start_engine())

bike.start_journey()
print(bike.start_engine())
```

Advanced OOP Concepts

6. Special Methods (Magic Methods/Dunder Methods)

- Methods with double underscores that define object behavior
- Enable operator overloading and custom object behavior
- Memory Tip: "DUNDER = Double UNDERscore" - Special powers!

```
class Money:
    def __init__(self, amount, currency="USD"):
        self.amount = amount
        self.currency = currency
```

```
# String representation for users
def __str__(self):
    return f"{self.amount} {self.currency}"

# String representation for developers
def __repr__(self):
    return f"Money({self.amount}, '{self.currency}')"

# Addition operator
def __add__(self, other):
    if isinstance(other, Money):
        if self.currency == other.currency:
            return Money(self.amount + other.amount, self.currency)
        else:
            raise ValueError("Cannot add different currencies")
    elif isinstance(other, (int, float)):
        return Money(self.amount + other, self.currency)
    else:
        return NotImplemented

# Subtraction operator
def __sub__(self, other):
    if isinstance(other, Money):
        if self.currency == other.currency:
            return Money(self.amount - other.amount, self.currency)
        else:
            raise ValueError("Cannot subtract different currencies")
    elif isinstance(other, (int, float)):
        return Money(self.amount - other, self.currency)
    else:
        return NotImplemented

# Multiplication operator
def __mul__(self, other):
    if isinstance(other, (int, float)):
        return Money(self.amount * other, self.currency)
    else:
        return NotImplemented

# Comparison operators
def __eq__(self, other):
    if isinstance(other, Money):
```

```
            return self.amount == other.amount and self.currency == other.currency
        return False

    def __lt__(self, other):
        if isinstance(other, Money) and self.currency == other.currency:
            return self.amount < other.amount
        return NotImplemented

    def __le__(self, other):
        return self < other or self == other

    # Length (for demonstration)
    def __len__(self):
        return len(str(self.amount))

    # Boolean conversion
    def __bool__(self):
        return self.amount != 0

    # Container-like behavior
    def __getitem__(self, key):
        if key == 0:
            return self.amount
        elif key == 1:
            return self.currency
        else:
            raise IndexError("Money index out of range")

# Usage of magic methods
money1 = Money(100)
money2 = Money(50)

print(money1)          # 100 USD (calls __str__)
print(repr(money1))    # Money(100, 'USD') (calls __repr__)

# Arithmetic operations
total = money1 + money2      # Calls __add__
difference = money1 - money2 # Calls __sub__
doubled = money1 * 2         # Calls __mul__

print(f"Total: {total}")
print(f"Difference: {difference}")
```

```
print(f"Doubled: {doubled}")

# Comparison operations
print(money1 == money2)     # False (calls __eq__)
print(money1 > money2)      # True (calls __lt__ indirectly)

# Other magic methods
print(len(money1))          # 3 (calls __len__)
print(bool(Money(0)))       # False (calls __bool__)
print(money1[0], money1[1])  # 100 USD (calls __getitem__)
```

7. Class Methods, Static Methods, and Instance Methods

- Instance methods: Work with instance data (self parameter)
- Class methods: Work with class data (@classmethod decorator)
- Static methods: Utility functions (@staticmethod decorator)
- Memory Tip: "ISC" - Instance (self), Static (no special param), Class (cls)

```
class Employee:
    # Class attributes
    company_name = "TechCorp"
    employee_count = 0
    min_salary = 30000

    def __init__(self, name, salary, department):
        # Instance attributes
        self.name = name
        self.salary = salary
        self.department = department
        self.employee_id = Employee.employee_count + 1

        # Update class attribute
        Employee.employee_count += 1

    # Instance method (works with instance data)
    def get_details(self):
        return f"ID: {self.employee_id}, Name: {self.name}, Salary: ${self.salary}"

    def give_raise(self, amount):
        self.salary += amount
        return f"{self.name} received a raise of ${amount}"

    # Class method (works with class data)
```

```
    @classmethod
    def get_company_info(cls):
        return f"Company: {cls.company_name}, Employees: {cls.employee_count}"

    @classmethod
    def set_company_name(cls, name):
        cls.company_name = name

    @classmethod
    def from_string(cls, emp_string):
        """Alternative constructor"""
        name, salary, department = emp_string.split('-')
        return cls(name, int(salary), department)

    # Static method (utility function, no access to instance or class)
    @staticmethod
    def is_valid_salary(salary):
        return salary >= Employee.min_salary

    @staticmethod
    def calculate_tax(salary):
        if salary < 50000:
            return salary * 0.1
        elif salary < 100000:
            return salary * 0.2
        else:
            return salary * 0.3

    @staticmethod
    def format_currency(amount):
        return f"${amount:,.2f}"

# Usage examples
# Instance methods
emp1 = Employee("Alice", 60000, "Engineering")
emp2 = Employee("Bob", 45000, "Marketing")

print(emp1.get_details())  # Instance method
print(emp1.give_raise(5000))  # Instance method

# Class methods
print(Employee.get_company_info())  # Class method
```

```
Employee.set_company_name("NewTechCorp")  # Class method

# Alternative constructor using class method
emp3 = Employee.from_string("Charlie-70000-Sales")
print(emp3.get_details())

# Static methods
print(Employee.is_valid_salary(25000))  # False
print(Employee.calculate_tax(60000))    # 12000.0
print(Employee.format_currency(75500))  # $75,500.00

# Static methods can be called from instances too
print(emp1.format_currency(emp1.salary))  # $65,000.00
```

8. Property Decorators and Descriptors

- Properties: Control access to attributes with getter, setter, deleter
- Descriptors: Objects that define attribute access behavior
- Memory Tip: "GSD" - Getter, Setter, Deleter for properties

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    # Property getter
    @property
    def celsius(self):
        return self._celsius

    # Property setter with validation
    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero!")
        self._celsius = value

    # Property deleter
    @celsius.deleter
    def celsius(self):
        print("Temperature data deleted")
        self._celsius = 0

    # Read-only property (no setter)
```

```
    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

    @property
    def kelvin(self):
        return self._celsius + 273.15

    # Another property with setter
    @property
    def fahrenheit_rw(self):
        return (self._celsius * 9/5) + 32

    @fahrenheit_rw.setter
    def fahrenheit_rw(self, value):
        self._celsius = (value - 32) * 5/9

# Custom descriptor
class ValidatedAttribute:
    def __init__(self, min_value=None, max_value=None):
        self.min_value = min_value
        self.max_value = max_value

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name, 0)

    def __set__(self, obj, value):
        if self.min_value is not None and value < self.min_value:
            raise ValueError(f"Value must be >= {self.min_value}")
        if self.max_value is not None and value > self.max_value:
            raise ValueError(f"Value must be <= {self.max_value}")
        obj.__dict__[self.name] = value

    def __set_name__(self, owner, name):
        self.name = name

class Student:
    age = ValidatedAttribute(min_value=0, max_value=120)
    grade = ValidatedAttribute(min_value=0, max_value=100)
```

```
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age      # Uses descriptor
        self.grade = grade  # Uses descriptor

# Usage of properties
temp = Temperature(25)
print(f"Celsius: {temp.celsius}")     # 25
print(f"Fahrenheit: {temp.fahrenheit}") # 77.0
print(f"Kelvin: {temp.kelvin}")       # 298.15

# Setting temperature
temp.celsius = 30
print(f"New Fahrenheit: {temp.fahrenheit}") # 86.0

# Setting via Fahrenheit
temp.fahrenheit_rw = 100
print(f"New Celsius: {temp.celsius}")  # 37.78

# Usage of descriptors
student = Student("Alice", 20, 95)
print(f"Age: {student.age}, Grade: {student.grade}")

# Validation in action
try:
    student.age = -5  # Error!
except ValueError as e:
    print(f"Error: {e}")
```

Design Patterns and Best Practices

9. Composition vs Inheritance

- Inheritance: "IS-A" relationship
- Composition: "HAS-A" relationship
- Memory Tip: "Composition over Inheritance" - Favor HAS-A over IS-A

```
# Inheritance approach (IS-A relationship)
class Animal:
    def __init__(self, name):
        self.name = name
```

```
    def make_sound(self):
        pass

class Dog(Animal):  # Dog IS-A Animal
    def make_sound(self):
        return f"{self.name} barks"

# Composition approach (HAS-A relationship)
class Engine:
    def __init__(self, horsepower, fuel_type):
        self.horsepower = horsepower
        self.fuel_type = fuel_type
        self.is_running = False

    def start(self):
        self.is_running = True
        return f"Engine started ({self.horsepower}HP, {self.fuel_type})"

    def stop(self):
        self.is_running = False
        return "Engine stopped"

class GPS:
    def __init__(self):
        self.current_location = "Unknown"

    def get_directions(self, destination):
        return f"Directions to {destination} from {self.current_location}"

class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        # Composition: Car HAS-A Engine and HAS-A GPS
        self.engine = Engine(200, "Gasoline")
        self.gps = GPS()

    def start_car(self):
        return self.engine.start()

    def navigate_to(self, destination):
        return self.gps.get_directions(destination)
```

```
# Usage
car = Car("Toyota", "Camry")
print(car.start_car())  # Engine started (200HP, Gasoline)
print(car.navigate_to("Mall"))  # Directions to Mall from Unknown
```

10. Singleton Pattern and Class Design

- Singleton: Ensures only one instance of a class exists
- Factory Pattern: Creates objects without specifying exact classes
- Memory Tip: "One SINGLE instance, FACTORY creates objects"

```
# Singleton Pattern
class DatabaseConnection:
    _instance = None
    _initialized = False

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not self._initialized:
            self.connection_string = "database://localhost:5432"
            self.is_connected = False
            DatabaseConnection._initialized = True

    def connect(self):
        self.is_connected = True
        return "Connected to database"

    def disconnect(self):
        self.is_connected = False
        return "Disconnected from database"

# Factory Pattern
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
```

```
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type.lower() == "dog":
            return Dog()
        elif animal_type.lower() == "cat":
            return Cat()
        else:
            raise ValueError(f"Unknown animal type: {animal_type}")

# Usage
# Singleton - multiple variables point to same instance
db1 = DatabaseConnection()
db2 = DatabaseConnection()
print(db1 is db2)  # True - same instance

# Factory - creates different types of objects
factory = AnimalFactory()
dog = factory.create_animal("dog")
cat = factory.create_animal("cat")
print(dog.make_sound())  # Woof!
print(cat.make_sound())  # Meow!
```

Real-World Applications and Examples

11. Banking System - Complete Example

```
from datetime import datetime
from abc import ABC, abstractmethod

class Account(ABC):
    account_counter = 1000

    def __init__(self, account_holder, initial_balance=0):
        self.account_number = Account.account_counter
```

```
    Account.account_counter += 1
    self.account_holder = account_holder
    self._balance = initial_balance
    self.transaction_history = []
    self.created_date = datetime.now()

@property
def balance(self):
    return self._balance

@abstractmethod
def calculate_interest(self):
    pass

def deposit(self, amount):
    if amount > 0:
        self._balance += amount
        self._add_transaction("DEPOSIT", amount)
        return f"Deposited ${amount}. New balance: ${self._balance}"
    return "Invalid deposit amount"

def withdraw(self, amount):
    if amount > 0 and amount <= self._balance:
        self._balance -= amount
        self._add_transaction("WITHDRAWAL", amount)
        return f"Withdrew ${amount}. New balance: ${self._balance}"
    return "Invalid withdrawal amount or insufficient funds"

def _add_transaction(self, transaction_type, amount):
    transaction = {
        'type': transaction_type,
        'amount': amount,
        'balance_after': self._balance,
        'timestamp': datetime.now()
    }
    self.transaction_history.append(transaction)

def get_statement(self):
    statement = f"\n--- Account Statement ---\n"
    statement += f"Account: {self.account_number}\n"
    statement += f"Holder: {self.account_holder}\n"
    statement += f"Current Balance: ${self._balance}\n"
```

```
        statement += f"Transactions:\n"

        for trans in self.transaction_history[-5:]:  # Last 5 transactions
            statement += f"  {trans['timestamp'].strftime('%Y-%m-%d %H:%M')} - "
            statement += f"{trans['type']}: ${trans['amount']}, "
            statement += f"Balance: ${trans['balance_after']}\n"

        return statement

class SavingsAccount(Account):
    def __init__(self, account_holder, initial_balance=0, interest_rate=0.02):
        super().__init__(account_holder, initial_balance)
        self.interest_rate = interest_rate
        self.account_type = "SAVINGS"

    def calculate_interest(self):
        interest = self._balance * self.interest_rate
        self._balance += interest
        self._add_transaction("INTEREST", interest)
        return f"Interest added: ${interest:.2f}"

class CheckingAccount(Account):
    def __init__(self, account_holder, initial_balance=0, overdraft_limit=500):
        super().__init__(account_holder, initial_balance)
        self.overdraft_limit = overdraft_limit
        self.account_type = "CHECKING"

    def calculate_interest(self):
        return "No interest for checking accounts"

    def withdraw(self, amount):
        if amount > 0 and amount <= (self._balance + self.overdraft_limit):
            self._balance -= amount
            self._add_transaction("WITHDRAWAL", amount)
            if self._balance < 0:
                return f"Withdrew ${amount}. Balance: ${self._balance} (Overdraft used)"
            return f"Withdrew ${amount}. New balance: ${self._balance}"
        return "Withdrawal exceeds overdraft limit"

class Bank:
    def __init__(self, name):
        self.name = name
```

```
        self.accounts = {}
        self.customers = {}

    def create_account(self, customer_name, account_type, initial_balance=0):
        if account_type.upper() == "SAVINGS":
            account = SavingsAccount(customer_name, initial_balance)
        elif account_type.upper() == "CHECKING":
            account = CheckingAccount(customer_name, initial_balance)
        else:
            return "Invalid account type"

        self.accounts[account.account_number] = account
        if customer_name not in self.customers:
            self.customers[customer_name] = []
        self.customers[customer_name].append(account.account_number)

        return f"Account {account.account_number} created for {customer_name}"

    def get_account(self, account_number):
        return self.accounts.get(account_number)

    def transfer_money(self, from_account_num, to_account_num, amount):
        from_account = self.get_account(from_account_num)
        to_account = self.get_account(to_account_num)

        if not from_account or not to_account:
            return "Invalid account number(s)"

        if from_account.balance >= amount:
            from_account.withdraw(amount)
            to_account.deposit(amount)
            return f"Transferred ${amount} from {from_account_num} to {to_account_num}"
        return "Insufficient funds for transfer"

# Usage Example
bank = Bank("TechBank")

# Create accounts
print(bank.create_account("Alice Johnson", "SAVINGS", 1000))
print(bank.create_account("Bob Smith", "CHECKING", 500))

# Get account objects
```

```
alice_account = bank.get_account(1000)
bob_account = bank.get_account(1001)

# Perform transactions
print(alice_account.deposit(500))
print(alice_account.calculate_interest())
print(bob_account.withdraw(600))  # Uses overdraft

# Transfer money
print(bank.transfer_money(1000, 1001, 200))

# Print statements
print(alice_account.get_statement())
print(bob_account.get_statement())
```

12. Game Development - RPG Character System

```
from enum import Enum
from random import randint

class CharacterClass(Enum):
    WARRIOR = "Warrior"
    MAGE = "Mage"
    ARCHER = "Archer"
    ROGUE = "Rogue"

class Weapon:
    def __init__(self, name, damage, weapon_type):
        self.name = name
        self.damage = damage
        self.weapon_type = weapon_type

    def __str__(self):
        return f"{self.name} (Damage: {self.damage})"

class Character:
    def __init__(self, name, character_class):
        self.name = name
        self.character_class = character_class
        self.level = 1
        self.experience = 0
        self.max_health = self._calculate_base_health()
```

```
        self.current_health = self.max_health
        self.max_mana = self._calculate_base_mana()
        self.current_mana = self.max_mana
        self.weapon = None
        self.skills = []
        self._set_class_attributes()

    def _calculate_base_health(self):
        base_health = {
            CharacterClass.WARRIOR: 120,
            CharacterClass.MAGE: 80,
            CharacterClass.ARCHER: 100,
            CharacterClass.ROGUE: 90
        }
        return base_health[self.character_class]

    def _calculate_base_mana(self):
        base_mana = {
            CharacterClass.WARRIOR: 30,
            CharacterClass.MAGE: 150,
            CharacterClass.ARCHER: 50,
            CharacterClass.ROGUE: 70
        }
        return base_mana[self.character_class]

    def _set_class_attributes(self):
        if self.character_class == CharacterClass.WARRIOR:
            self.strength = 18
            self.intelligence = 8
            self.agility = 10
        elif self.character_class == CharacterClass.MAGE:
            self.strength = 6
            self.intelligence = 20
            self.agility = 8
        elif self.character_class == CharacterClass.ARCHER:
            self.strength = 12
            self.intelligence = 10
            self.agility = 18
        elif self.character_class == CharacterClass.ROGUE:
            self.strength = 10
            self.intelligence = 12
            self.agility = 16
```

```
    def equip_weapon(self, weapon):
        self.weapon = weapon
        return f"{self.name} equipped {weapon.name}"

    def attack(self, target):
        base_damage = self.strength * 2
        weapon_damage = self.weapon.damage if self.weapon else 0
        total_damage = base_damage + weapon_damage + randint(1, 10)

        # Critical hit chance
        if randint(1, 100) <= self.agility:
            total_damage *= 2
            critical = " (Critical Hit!)"
        else:
            critical = ""

        target.take_damage(total_damage)
        return f"{self.name} attacks {target.name} for {total_damage} damage{critical}"

    def take_damage(self, damage):
        self.current_health = max(0, self.current_health - damage)
        if self.current_health == 0:
            return f"{self.name} has been defeated!"
        return f"{self.name} takes {damage} damage. Health:
{self.current_health}/{self.max_health}"

    def heal(self, amount):
        old_health = self.current_health
        self.current_health = min(self.max_health, self.current_health + amount)
        healed = self.current_health - old_health
        return f"{self.name} healed for {healed} HP. Health:
{self.current_health}/{self.max_health}"

    def gain_experience(self, exp):
        self.experience += exp
        if self.experience >= self.level * 100:
            return self.level_up()
        return f"{self.name} gained {exp} experience"

    def level_up(self):
        self.level += 1
```

```
        self.experience = 0

        # Increase stats
        health_increase = 20
        mana_increase = 15
        self.max_health += health_increase
        self.current_health = self.max_health
        self.max_mana += mana_increase
        self.current_mana = self.max_mana

        # Increase class-specific attributes
        if self.character_class == CharacterClass.WARRIOR:
            self.strength += 3
            self.agility += 1
        elif self.character_class == CharacterClass.MAGE:
            self.intelligence += 3
            self.strength += 1
        elif self.character_class == CharacterClass.ARCHER:
            self.agility += 3
            self.intelligence += 1
        elif self.character_class == CharacterClass.ROGUE:
            self.agility += 2
            self.intelligence += 2

        return f"{self.name} leveled up to {self.level}! All stats increased!"

    def get_stats(self):
        return f"""
--- {self.name} ({self.character_class.value}) Level {self.level} ---
Health: {self.current_health}/{self.max_health}
Mana: {self.current_mana}/{self.max_mana}
Strength: {self.strength}
Intelligence: {self.intelligence}
Agility: {self.agility}
Experience: {self.experience}/{self.level * 100}
Weapon: {self.weapon if self.weapon else "None"}
        """

# Usage Example
warrior = Character("Conan", CharacterClass.WARRIOR)
mage = Character("Gandalf", CharacterClass.MAGE)
```

```
# Create weapons
sword = Weapon("Iron Sword", 25, "Melee")
staff = Weapon("Fire Staff", 30, "Magic")

# Equip weapons
print(warrior.equip_weapon(sword))
print(mage.equip_weapon(staff))

# Battle simulation
print("\n--- Battle Begins ---")
print(warrior.attack(mage))
print(mage.take_damage(0))  # Damage already applied in attack
print(mage.attack(warrior))
print(warrior.take_damage(0))

# Healing and experience
print(warrior.heal(20))
print(warrior.gain_experience(150))

# Display stats
print(warrior.get_stats())
print(mage.get_stats())
```

13. E-commerce System - Product Management

```
from datetime import datetime
from collections import defaultdict

class Product:
    def __init__(self, product_id, name, price, category, stock_quantity):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.category = category
        self.stock_quantity = stock_quantity
        self.created_date = datetime.now()
        self.reviews = []

    def add_review(self, customer_name, rating, comment):
        if 1 <= rating <= 5:
            review = {
                'customer': customer_name,
```

```
                'rating': rating,
                'comment': comment,
                'date': datetime.now()
            }
            self.reviews.append(review)
            return "Review added successfully"
        return "Invalid rating. Must be between 1 and 5"

    def get_average_rating(self):
        if not self.reviews:
            return 0
        return sum(review['rating'] for review in self.reviews) / len(self.reviews)

    def update_stock(self, quantity_change):
        new_quantity = self.stock_quantity + quantity_change
        if new_quantity >= 0:
            self.stock_quantity = new_quantity
            return f"Stock updated. New quantity: {self.stock_quantity}"
        return "Insufficient stock for this operation"

    def __str__(self):
        return f"{self.name} - ${self.price} (Stock: {self.stock_quantity})"

class Customer:
    def __init__(self, customer_id, name, email):
        self.customer_id = customer_id
        self.name = name
        self.email = email
        self.order_history = []
        self.cart = ShoppingCart()

    def add_to_cart(self, product, quantity):
        return self.cart.add_item(product, quantity)

    def remove_from_cart(self, product_id):
        return self.cart.remove_item(product_id)

    def place_order(self):
        if self.cart.items:
            order = Order(len(self.order_history) + 1, self)
            for item in self.cart.items:
                order.add_item(item['product'], item['quantity'])
```

```
            # Check stock availability
            if order.validate_stock():
                # Reduce stock
                for item in order.items:
                    item['product'].update_stock(-item['quantity'])

                self.order_history.append(order)
                self.cart.clear_cart()
                return f"Order {order.order_id} placed successfully!"
            else:
                return "Order failed: Insufficient stock for some items"
        return "Cart is empty"

class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, product, quantity):
        # Check if product already in cart
        for item in self.items:
            if item['product'].product_id == product.product_id:
                item['quantity'] += quantity
                return f"Updated {product.name} quantity to {item['quantity']}"

        # Add new item
        self.items.append({'product': product, 'quantity': quantity})
        return f"Added {quantity} {product.name} to cart"

    def remove_item(self, product_id):
        self.items = [item for item in self.items if item['product'].product_id != product_id]
        return "Item removed from cart"

    def get_total(self):
        return sum(item['product'].price * item['quantity'] for item in self.items)

    def clear_cart(self):
        self.items = []

    def __str__(self):
        if not self.items:
            return "Cart is empty"
```

```
        cart_str = "Shopping Cart:\n"
        for item in self.items:
            cart_str += f"  {item['product'].name} x{item['quantity']} = ${item['product'].price *
item['quantity']}\n"
        cart_str += f"Total: ${self.get_total()}"
        return cart_str

class Order:
    def __init__(self, order_id, customer):
        self.order_id = order_id
        self.customer = customer
        self.items = []
        self.order_date = datetime.now()
        self.status = "Pending"
        self.total_amount = 0

    def add_item(self, product, quantity):
        self.items.append({'product': product, 'quantity': quantity})
        self.total_amount += product.price * quantity

    def validate_stock(self):
        for item in self.items:
            if item['product'].stock_quantity < item['quantity']:
                return False
        return True

    def update_status(self, new_status):
        valid_statuses = ["Pending", "Processing", "Shipped", "Delivered", "Cancelled"]
        if new_status in valid_statuses:
            self.status = new_status
            return f"Order {self.order_id} status updated to {new_status}"
        return "Invalid status"

    def __str__(self):
        order_str = f"Order #{self.order_id} - {self.customer.name}\n"
        order_str += f"Date: {self.order_date.strftime('%Y-%m-%d %H:%M')}\n"
        order_str += f"Status: {self.status}\n"
        order_str += "Items:\n"
        for item in self.items:
            order_str += f"  {item['product'].name} x{item['quantity']} = ${item['product'].price
* item['quantity']}\n"
```

```
        order_str += f"Total: ${self.total_amount}"
        return order_str

class ECommerceStore:
    def __init__(self, store_name):
        self.store_name = store_name
        self.products = {}
        self.customers = {}
        self.orders = {}
        self.categories = defaultdict(list)

    def add_product(self, product):
        self.products[product.product_id] = product
        self.categories[product.category].append(product)
        return f"Product {product.name} added to store"

    def add_customer(self, customer):
        self.customers[customer.customer_id] = customer
        return f"Customer {customer.name} registered"

    def search_products(self, keyword):
        results = []
        for product in self.products.values():
            if keyword.lower() in product.name.lower() or keyword.lower() in
product.category.lower():
                results.append(product)
        return results

    def get_products_by_category(self, category):
        return self.categories.get(category, [])

    def get_top_rated_products(self, limit=5):
        products_with_ratings = [(p, p.get_average_rating()) for p in self.products.values() if
p.reviews]
        products_with_ratings.sort(key=lambda x: x[1], reverse=True)
        return [p[0] for p in products_with_ratings[:limit]]

    def generate_sales_report(self):
        total_orders = len(self.orders)
        total_revenue = sum(order.total_amount for order in self.orders.values())

        category_sales = defaultdict(int)
```

```
        for order in self.orders.values():
            for item in order.items:
                category_sales[item['product'].category] += item['quantity']

        report = f"\n--- Sales Report for {self.store_name} ---\n"
        report += f"Total Orders: {total_orders}\n"
        report += f"Total Revenue: ${total_revenue}\n"
        report += f"Sales by Category:\n"
        for category, quantity in category_sales.items():
            report += f"  {category}: {quantity} units\n"

        return report

# Usage Example
store = ECommerceStore("TechStore")

# Add products
laptop = Product(1, "Gaming Laptop", 1200, "Electronics", 10)
mouse = Product(2, "Wireless Mouse", 25, "Electronics", 50)
keyboard = Product(3, "Mechanical Keyboard", 80, "Electronics", 30)

store.add_product(laptop)
store.add_product(mouse)
store.add_product(keyboard)

# Add customer
customer = Customer(1, "John Doe", "john@email.com")
store.add_customer(customer)

# Shopping process
print(customer.add_to_cart(laptop, 1))
print(customer.add_to_cart(mouse, 2))
print(customer.cart)

# Place order
result = customer.place_order()
print(result)

# Add reviews
laptop.add_review("John Doe", 5, "Excellent laptop!")
mouse.add_review("John Doe", 4, "Good mouse, works well")
```

```
# Search products
results = store.search_products("laptop")
print(f"Search results: {[str(p) for p in results]}")

# Generate report
print(store.generate_sales_report())
```

Quick Master Reference Guide

The Big Four Principles (Remember: EIAP)

1. Encapsulation - Bundle data and methods, control access
2. Inheritance - Reuse code through parent-child relationships
3. Abstraction - Hide implementation details, show interface
4. Polymorphism - Same interface, different implementations

Method Types (Remember: ISC)

- Instance methods - def method(self) - work with instance data
- Static methods - @staticmethod - utility functions, no special parameters
- Class methods - @classmethod - work with class data, take cls parameter

Access Modifiers (Remember: PPP)

- Public - attribute - accessible everywhere
- Protected - _attribute - internal use (convention)
- Private - __attribute - name mangling, truly private

Magic Methods (Remember: SRAC)

- String representation - __str__, __repr__
- Right-hand operations - __add__, __sub__, __mul__
- Access control - __getitem__, __setitem__
- Comparison - __eq__, __lt__, __gt__

Property Pattern (Remember: GSD)

```
@property
def attribute(self):  # G - Getter
   return self._attribute

@attribute.setter
def attribute(self, value):  # S - Setter
```

```
    self._attribute = value

@attribute.deleter
def attribute(self):  # D - Deleter
    del self._attribute
```

Inheritance Best Practices

- Use super() to call parent methods
- Override methods for specialized behavior
- Composition over inheritance when possible
- Abstract base classes for interfaces

Common Exam Patterns

1. Bank Account System - Encapsulation with balance protection
2. Shape Hierarchy - Inheritance with area/perimeter calculations
3. Vehicle System - Polymorphism with different vehicle types
4. Student Management - Multiple classes with relationships
5. Game Characters - Complex inheritance with stats and abilities

Remember: OOP is about modeling real-world relationships in code. Think in terms of objects, their properties, behaviors, and how they interact with each other!

========== Write your notes below==========

Python Built-in Functions - Complete Reference Guide

Definition

Built-in functions in Python are pre-defined functions that are always available without importing any module. They provide essential functionality for data manipulation, type conversion, input/output operations, mathematical calculations, and object introspection. Python has 69 built-in functions that cover all fundamental programming needs.

A - Arithmetic and Mathematical Functions

abs() - Returns absolute value of a number

divmod() - Returns quotient and remainder of division as tuple

max() - Returns the largest item in an iterable or largest of arguments

min() - Returns the smallest item in an iterable or smallest of arguments

pow() - Returns base raised to power, optionally modulo a third argument

round() - Returns floating point number rounded to given precision

sum() - Returns sum of all items in an iterable

B - Boolean and Logical Functions

all() - Returns True if all elements in iterable are true

any() - Returns True if any element in iterable is true

bool() - Returns boolean value of an object

C - Collection and Container Functions

dict() - Creates a dictionary object

frozenset() - Creates an immutable set object

list() - Creates a list object

set() - Creates a set object

tuple() - Creates a tuple object

D - Data Type Checking and Conversion

bin() - Converts integer to binary string with '0b' prefix

hex() - Converts integer to hexadecimal string with '0x' prefix

oct() - Converts integer to octal string with '0o' prefix

chr() - Returns character corresponding to ASCII/Unicode code

ord() - Returns ASCII/Unicode code of a character

complex() - Creates a complex number

float() - Converts value to floating point number

int() - Converts value to integer

str() - Converts object to string representation

bytes() - Creates bytes object from string, integers, or iterable

bytearray() - Creates mutable array of bytes

E - Enumeration and Iteration Functions

enumerate() - Returns enumerate object with index-value pairs

iter() - Returns iterator object from an iterable

next() - Returns next item from iterator

range() - Generates sequence of numbers

reversed() - Returns reversed iterator of a sequence

zip() - Combines multiple iterables into tuples

F - Filtering and Mapping Functions

filter() - Filters elements from iterable using function

map() - Applies function to every item in iterable

sorted() - Returns sorted list from elements of any iterable

G - General Purpose and Utility Functions

callable() - Checks if object is callable (function, method, class)

hash() - Returns hash value of an object

id() - Returns unique identity of an object

isinstance() - Checks if object is instance of specified class

issubclass() - Checks if class is subclass of another class

len() - Returns length/count of items in object

repr() - Returns string representation of object for developers

type() - Returns type of an object or creates new type

H - Help and Documentation Functions

help() - Displays help documentation for objects

dir() - Returns list of attributes and methods of an object

I - Input/Output Functions

input() - Reads string input from user

print() - Outputs values to console or file

open() - Opens file and returns file object

J - Object Attribute Functions

delattr() - Deletes attribute from an object

getattr() - Gets value of named attribute from object

hasattr() - Checks if object has specified attribute

setattr() - Sets value of named attribute of an object

vars() - Returns dict attribute of an object

K - Code Execution Functions

compile() - Compiles source code into code object

eval() - Evaluates string expression and returns result

exec() - Executes Python code dynamically

globals() - Returns dictionary of global symbol table

locals() - Returns dictionary of local symbol table

L - Advanced Object Functions

classmethod() - Converts method to class method

staticmethod() - Converts method to static method

property() - Creates property object for class attributes

super() - Returns proxy object for accessing parent class methods

---

M - Memory and Format Functions

format() - Formats value according to format specification

memoryview() - Returns memory view object of given argument

object() - Returns new featureless object (base of all classes)

slice() - Returns slice object for slicing sequences

Functions by Category - Quick Reference

Type Conversion Functions

- int() - Convert to integer
- float() - Convert to float
- str() - Convert to string
- bool() - Convert to boolean
- complex() - Convert to complex number
- list() - Convert to list
- tuple() - Convert to tuple
- set() - Convert to set
- dict() - Convert to dictionary
- bytes() - Convert to bytes
- bytearray() - Convert to byte array

Mathematical Functions

- abs() - Absolute value
- divmod() - Division with remainder
- max() - Maximum value
- min() - Minimum value
- pow() - Power operation
- round() - Round number
- sum() - Sum of iterable

Sequence Functions

- len() - Length of sequence
- sorted() - Sort sequence

- reversed() - Reverse sequence
- enumerate() - Add indices to sequence
- zip() - Combine sequences
- range() - Generate number sequence

Logical Functions

- all() - All elements true
- any() - Any element true
- bool() - Boolean conversion

Object Inspection Functions

- type() - Object type
- isinstance() - Check instance
- issubclass() - Check subclass
- hasattr() - Check attribute
- getattr() - Get attribute
- setattr() - Set attribute
- delattr() - Delete attribute
- dir() - List attributes
- vars() - Object dictionary
- id() - Object identity
- callable() - Check if callable

Functional Programming Functions

- map() - Apply function to iterable
- filter() - Filter iterable
- iter() - Create iterator
- next() - Get next item

String and Character Functions

- chr() - ASCII/Unicode to character
- ord() - Character to ASCII/Unicode
- repr() - Developer string representation
- format() - Format string

Number Base Functions

- bin() - Convert to binary
- hex() - Convert to hexadecimal
- oct() - Convert to octal

Code Execution Functions

- eval() - Evaluate expression
- exec() - Execute code
- compile() - Compile code

Scope Functions

- globals() - Global variables
- locals() - Local variables

Advanced Object Functions

- super() - Parent class access
- classmethod() - Class method decorator
- staticmethod() - Static method decorator
- property() - Property decorator

I/O Functions

- input() - User input
- print() - Output to console
- open() - File operations

Memory Functions

- hash() - Object hash
- memoryview() - Memory view
- object() - Base object

Help Functions

- help() - Documentation
- dir() - Object attributes

Functions by Data Types

Numeric Types (int, float, complex)

- abs() - Absolute value
- bin() - Binary representation
- bool() - Boolean conversion
- complex() - Complex number creation
- divmod() - Division with remainder

- float() - Float conversion
- hex() - Hexadecimal representation
- int() - Integer conversion
- oct() - Octal representation
- pow() - Power calculation
- round() - Rounding

String Type (str)

- str() - String conversion
- chr() - Character from code
- ord() - Code from character
- format() - String formatting
- repr() - String representation

Sequence Types (list, tuple, range)

- list() - List creation
- tuple() - Tuple creation
- range() - Range creation
- len() - Length
- max() - Maximum
- min() - Minimum
- sorted() - Sorting
- reversed() - Reversing
- sum() - Summation
- enumerate() - Enumeration
- zip() - Zipping

Set Types (set, frozenset)

- set() - Set creation
- frozenset() - Frozen set creation

Mapping Type (dict)

- dict() - Dictionary creation

Binary Types (bytes, bytearray)

- bytes() - Bytes creation
- bytearray() - Byte array creation
- memoryview() - Memory view

Iterator Types

- iter() - Iterator creation
- next() - Next item

Functions by Usage Context

Beginner Level Functions

- print() - Output
- input() - Input
- len() - Length
- type() - Type checking
- str(), int(), float() - Basic conversions
- max(), min() - Extremes
- sum() - Addition
- range() - Number sequences

Intermediate Level Functions

- enumerate() - Indexed iteration
- zip() - Parallel iteration
- sorted() - Sorting
- reversed() - Reversing
- all(), any() - Boolean operations
- isinstance() - Type checking
- hasattr(), getattr(), setattr() - Attribute operations
- map(), filter() - Functional programming

Advanced Level Functions

- eval(), exec() - Code execution
- compile() - Code compilation
- globals(), locals() - Scope inspection
- super() - Inheritance
- classmethod(), staticmethod(), property() - Decorators
- memoryview() - Memory operations
- slice() - Advanced slicing

Memory Tips for Function Categories

"MATH-SLZ" - Mathematical Functions

- Max, Abs, Type conversion, Hex/bin/oct, Sum, Len, Zip

"BITE" - Boolean and Iteration Functions

- Bool/all/any, Iter/next, Type checking, Enumerate

"FAIR" - Functional and Advanced Functions

- Filter/map, Attribute functions, Inspection functions, Repr/format

"SCOPE" - System and Code Functions

- Super/static/class methods, Compile/eval/exec, Open/input/print, Property, Error handling

---

Function Return Types Quick Reference

Returns Numbers

- abs() → int/float
- len() → int
- max(), min() → same as input
- sum() → int/float
- hash() → int
- id() → int
- ord() → int

Returns Strings

- str() → str
- repr() → str
- bin(), hex(), oct() → str
- chr() → str
- input() → str

Returns Boolean

- bool() → bool
- all(), any() → bool

- isinstance(), issubclass() → bool
- hasattr() → bool
- callable() → bool

Returns Iterators/Objects

- iter() → iterator
- enumerate() → enumerate object
- zip() → zip object
- map() → map object
- filter() → filter object
- reversed() → reverse iterator
- range() → range object

Returns Collections

- list() → list
- tuple() → tuple
- set() → set
- dict() → dict
- sorted() → list

Remember: These 69 built-in functions form the core toolkit of Python programming. Master these functions and you'll have the foundation for solving any programming problem efficiently!

25 Amazing Python Project Ideas Using Core Python

🟢 BASIC LEVEL (Projects 1-8) - Core Python Fundamentals

1. Simple Calculator

What it does: Basic math operations (+, -, *, /, %) **Core concepts:** Variables, input/output, functions, if-else statements **Features:** Add, subtract, multiply, divide two numbers **Lines of code:** ~30-40

2. Number Guessing Game

What it does: Computer picks a random number, user guesses **Core concepts:** Random module, loops, conditionals, input validation **Features:** 1-100 range, guess counter, high/low hints **Lines of code:** ~25-35

3. Password Generator

What it does: Creates random passwords with different criteria **Core concepts:** Random module, strings, loops, lists **Features:** Set length, include/exclude special characters **Lines of code:** ~40-50

4. Word Counter

What it does: Counts words, characters, and lines in text **Core concepts:** String methods, file reading, dictionaries **Features:** Read from file or user input, basic statistics **Lines of code:** ~35-45

5. Temperature Converter

What it does: Converts between Celsius, Fahrenheit, and Kelvin **Core concepts:** Functions, mathematical operations, input validation **Features:** All temperature scale conversions, user-friendly interface **Lines of code:** ~40-50

6. Simple To-Do List

What it does: Add, remove, and view tasks **Core concepts:** Lists, functions, basic file operations **Features:** Add tasks, mark complete, save to file **Lines of code:** ~60-70

7. Rock Paper Scissors

What it does: Classic game against computer **Core concepts:** Random choice, conditionals, loops, score tracking **Features:** Best of 3/5, score tracking, play again option **Lines of code:** ~50-60

8. Unit Converter

What it does: Convert between different units (length, weight, volume) **Core concepts:** Functions, dictionaries, mathematical operations **Features:** Multiple unit types, conversion factors **Lines of code:** ~70-80

🟡 INTERMEDIATE LEVEL (Projects 9-17) - Data Structures & File Handling

9. Contact Book

What it does: Store and manage contacts with phone numbers and emails **Core concepts:** Dictionaries, file I/O, search algorithms, data validation **Features:** Add, edit, delete, search contacts, save to file **Lines of code:** ~100-120

10. Expense Tracker

What it does: Track daily expenses by category **Core concepts:** Lists, dictionaries, file handling, date operations **Features:** Add expenses, view by category, monthly summaries **Lines of code:** ~120-140

11. Hangman Game

What it does: Word guessing game with visual hangman **Core concepts:** Lists, strings, file reading, ASCII art, game logic **Features:** Word categories, hint system, ASCII drawings **Lines of code:** ~100-130

12. Simple Quiz App

What it does: Multiple choice quiz with scoring **Core concepts:** Lists of dictionaries, file handling, score calculation **Features:** Different topics, score tracking, question bank **Lines of code:** ~110-130

13. File Organizer

What it does: Organize files by extension into folders **Core concepts:** OS module, file operations, directory handling **Features:** Sort by file type, create folders, move files **Lines of code:** ~80-100

14. Student Grade Manager

What it does: Calculate grades and GPA for students **Core concepts:** Classes, dictionaries, mathematical operations, file I/O **Features:** Add students, record grades, calculate GPA, generate reports **Lines of code:** ~140-160

15. Simple Bank Account

What it does: Basic banking operations (deposit, withdraw, balance) **Core concepts:** Classes, file persistence, error handling, validation **features:** Account creation, transaction history, balance tracking **Lines of code:** ~120-150

16. Weather Log

What it does: Log daily weather data and show basic statistics **Core concepts:** File handling, date operations, basic statistics, data analysis **Features:** Daily entries, temperature trends, rainfall tracking **Lines of code:** ~130-150

17. Tic-Tac-Toe

What it does: Classic 3x3 grid game for two players **Core concepts:** 2D lists, game logic, input validation, win detection **Features:** Two player mode, win/draw detection, replay option **Lines of code:** ~120-140

🔴 ADVANCED LEVEL (Projects 18-25) - Complex Logic & Algorithms

18. Library Management System

What it does: Manage books, members, and borrowing system **Core concepts:** Classes, file handling, date calculations, search algorithms **Features:** Book catalog, member management, due date tracking, fines **Lines of code:** ~200-250

19. Snake Game (Console)

What it does: Classic snake game in terminal **Core concepts:** 2D arrays, game loops, collision detection, timing **Features:** Growing snake, food spawning, score system, game over **Lines of code:** ~180-220

20. Password Manager

What it does: Store and encrypt passwords securely **Core concepts:** Encryption, file security, classes, data validation **Features:** Master password, password generation, categories, search **Lines of code:** ~200-240

21. Inventory Management

What it does: Track products, stock levels, and sales **Core concepts:** Classes, file handling, data validation, reports **Features:** Product management, stock alerts, sales tracking, reports **Lines of code:** ~220-260

22. Text-Based Adventure Game

What it does: Interactive story game with choices and inventory **Core concepts:** Classes, dictionaries, file I/O, game state management **Features:** Multiple rooms, inventory system, save/load game **Lines of code:** ~250-300

23. Log File Analyzer

What it does: Parse and analyze server/application logs **Core concepts:** Regular expressions, file processing, data analysis **Features:** Error detection, pattern matching, statistics, reports **Lines of code:** ~200-250

24. Mini Database System

What it does: Simple database with tables, queries, and relationships **Core concepts:** File handling, data structures, search algorithms, indexing **Features:** Create tables, insert/update/delete records, simple queries **Lines of code:** ~300-350

25. Personal Finance Dashboard

What it does: Comprehensive financial tracking with budgets and goals **Core concepts:** Classes, file handling, data analysis, report generation **Features:** Multiple accounts, budget tracking, goal setting, visual reports **Lines of code:** ~280-320

📊 Skill Progression Guide

Basic Level Focus:

- Variables and data types
- Input/output operations
- Conditional statements
- Loops and functions
- Basic string and math operations

Intermediate Level Focus:

- Data structures (lists, dictionaries)
- File input/output
- Error handling
- Object-oriented programming basics
- Algorithm implementation

Advanced Level Focus:

- Complex class hierarchies
- Advanced file operations
- Data validation and security
- Algorithm optimization
- System integration

Key Learning Path:

1. **Start with projects 1-3** to build confidence
2. **Complete 2-3 projects per level** before advancing
3. **Focus on understanding concepts** over rushing through
4. **Add your own features** to make projects unique
5. **Practice code organization** and documentation

Time Estimates:

- **Basic projects:** 2-4 hours each
- **Intermediate projects:** 4-8 hours each
- **Advanced projects:** 8-15 hours each

Each project builds upon previous concepts while introducing new skills, ensuring steady progression from Python beginner to intermediate developer!

💡 Learning Path Recommendations

Phase 1: Foundation Building (Projects 1-8)

Start with simpler games and basic tools to understand core concepts

Phase 2: System Development (Projects 9-16)

Move to more complex systems requiring advanced planning and design

Phase 3: Specialization (Projects 17-25)

Focus on areas of interest like data analysis, creativity, or utility tools

---

Remember: Each project should be built incrementally, starting with basic functionality and gradually adding advanced features. Document your code, handle errors gracefully, and always think about user experience!

JARVIS AI virtual voice assistant

Here is the complete code of jarvis ai assistant using gemini-2.0-flash API key

```
import speech_recognition as sr
import pyttsx3
import google.generativeai as genai
import requests
import json
import datetime
import webbrowser
import os
import subprocess
import threading
import time
import wikipedia
import wolframalpha
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import pyautogui
import psutil
import cv2
import numpy as np
from PIL import Image
import io
import base64

class JarvisAI:
    def __init__(self, gemini_api_key, wolfram_api_key=None):
        # Initialize Gemini AI
        genai.configure(api_key=gemini_api_key)
        self.model = genai.GenerativeModel('gemini-2.0-flash-exp')

        # Initialize speech components
        self.recognizer = sr.Recognizer()
        self.microphone = sr.Microphone()
        self.tts_engine = pyttsx3.init()

        # Configure TTS
        voices = self.tts_engine.getProperty('voices')
```

```
        self.tts_engine.setProperty('voice', voices[0].id)  # Use first available voice
        self.tts_engine.setProperty('rate', 180)
        self.tts_engine.setProperty('volume', 0.9)

        # Wolfram Alpha client (optional)
        self.wolfram_client = wolframalpha.Client(wolfram_api_key) if wolfram_api_key else
None

        # System state
        self.listening = False
        self.running = True

        # Adjust for ambient noise
        print("Calibrating microphone for ambient noise...")
        with self.microphone as source:
            self.recognizer.adjust_for_ambient_noise(source)
        print("Microphone calibrated!")

    def speak(self, text):
        """Convert text to speech"""
        print(f"Jarvis: {text}")
        self.tts_engine.say(text)
        self.tts_engine.runAndWait()

    def listen(self):
        """Listen for voice input"""
        try:
            with self.microphone as source:
                print("Listening...")
                audio = self.recognizer.listen(source, timeout=5, phrase_time_limit=10)

            print("Processing speech...")
            text = self.recognizer.recognize_google(audio)
            print(f"You said: {text}")
            return text.lower()

        except sr.WaitTimeoutError:
            return None
        except sr.UnknownValueError:
            self.speak("I didn't catch that. Could you repeat?")
            return None
        except sr.RequestError as e:
```

```
            self.speak("Speech recognition service is unavailable")
            return None

    def get_gemini_response(self, prompt, image_data=None):
        """Get response from Gemini AI"""
        try:
            if image_data:
                # Handle image input
                response = self.model.generate_content([prompt, image_data])
            else:
                response = self.model.generate_content(prompt)
            return response.text
        except Exception as e:
            return f"Sorry, I encountered an error: {str(e)}"

    def capture_screenshot(self):
        """Capture screenshot for visual tasks"""
        screenshot = pyautogui.screenshot()
        img_buffer = io.BytesIO()
        screenshot.save(img_buffer, format='PNG')
        img_buffer.seek(0)
        return Image.open(img_buffer)

    def get_weather(self, city="your location"):
        """Get weather information"""
        try:
            # Using OpenWeatherMap API (you'll need to get a free API key)
            api_key = "YOUR_OPENWEATHER_API_KEY"  # Replace with your API key
            url =
f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=met
ric"
            response = requests.get(url)
            data = response.json()

            if data["cod"] == 200:
                temp = data["main"]["temp"]
                desc = data["weather"][0]["description"]
                return f"The temperature in {city} is {temp}°C with {desc}"
            else:
                return "I couldn't fetch the weather information"
        except:
            return "Weather service is currently unavailable"
```

```
def get_time(self):
    """Get current time"""
    now = datetime.datetime.now()
    return f"The current time is {now.strftime('%I:%M %p')}"

def get_date(self):
    """Get current date"""
    now = datetime.datetime.now()
    return f"Today is {now.strftime('%A, %B %d, %Y')}"

def search_web(self, query):
    """Open web search"""
    url = f"https://www.google.com/search?q={query.replace(' ', '+')}"
    webbrowser.open(url)
    return f"I've opened a web search for {query}"

def open_application(self, app_name):
    """Open applications"""
    apps = {
        'notepad': 'notepad.exe',
        'calculator': 'calc.exe',
        'paint': 'mspaint.exe',
        'browser': 'chrome.exe',
        'chrome': 'chrome.exe',
        'firefox': 'firefox.exe',
        'explorer': 'explorer.exe',
        'cmd': 'cmd.exe',
        'terminal': 'cmd.exe'
    }

    app_name = app_name.lower()
    if app_name in apps:
        try:
            subprocess.Popen(apps[app_name])
            return f"Opening {app_name}"
        except:
            return f"Couldn't open {app_name}"
    else:
        return f"I don't know how to open {app_name}"

def get_system_info(self):
```

```
        """Get system information"""
        cpu_percent = psutil.cpu_percent(interval=1)
        memory = psutil.virtual_memory()
        disk = psutil.disk_usage('/')

        return f"CPU usage: {cpu_percent}%, Memory usage: {memory.percent}%, Disk usage:
{disk.percent}%"

    def wikipedia_search(self, query):
        """Search Wikipedia"""
        try:
            summary = wikipedia.summary(query, sentences=2)
            return summary
        except wikipedia.exceptions.DisambiguationError as e:
            return f"Multiple results found. Could you be more specific? Options include: {',
'.join(e.options[:3])}"
        except wikipedia.exceptions.PageError:
            return f"No Wikipedia page found for {query}"
        except:
            return "Wikipedia search failed"

    def wolfram_query(self, query):
        """Query Wolfram Alpha for calculations and factual questions"""
        if not self.wolfram_client:
            return "Wolfram Alpha is not configured"

        try:
            res = self.wolfram_client.query(query)
            answer = next(res.results).text
            return answer
        except:
            return "I couldn't process that calculation"

    def send_email(self, to_email, subject, body, smtp_server="smtp.gmail.com",
smtp_port=587, email="", password=""):
        """Send email (requires email configuration)"""
        try:
            msg = MIMEMultipart()
            msg['From'] = email
            msg['To'] = to_email
            msg['Subject'] = subject
            msg.attach(MIMEText(body, 'plain'))
```

```
            server = smtplib.SMTP(smtp_server, smtp_port)
            server.starttls()
            server.login(email, password)
            server.send_message(msg)
            server.quit()

            return f"Email sent to {to_email}"
        except:
            return "Failed to send email. Please check your email configuration."

    def control_system(self, command):
        """Control system functions"""
        command = command.lower()

        if "shutdown" in command:
            self.speak("Shutting down the system in 10 seconds. Say cancel to abort.")
            # Add shutdown logic here
            return "System shutdown initiated"
        elif "restart" in command:
            self.speak("Restarting the system in 10 seconds. Say cancel to abort.")
            # Add restart logic here
            return "System restart initiated"
        elif "sleep" in command:
            os.system("rundll32.exe powrprof.dll,SetSuspendState 0,1,0")
            return "Putting system to sleep"
        elif "lock" in command:
            os.system("rundll32.exe user32.dll,LockWorkStation")
            return "Locking the workstation"
        else:
            return "Unknown system command"

    def process_command(self, command):
        """Process voice commands"""
        command = command.lower()

        # Wake word check
        if "jarvis" not in command and not self.listening:
            return None

        # Remove wake word
        command = command.replace("jarvis", "").strip()
```

```
        # Time and date commands
        if any(word in command for word in ["time", "clock"]):
            return self.get_time()

        elif any(word in command for word in ["date", "today"]):
            return self.get_date()

        # Weather commands
        elif "weather" in command:
            city = command.replace("weather", "").replace("in", "").strip()
            return self.get_weather(city if city else "your location")

        # Web search commands
        elif any(word in command for word in ["search", "google", "look up"]):
            query = command.replace("search", "").replace("google", "").replace("look up",
"").strip()
            return self.search_web(query)

        # Application commands
        elif "open" in command:
            app = command.replace("open", "").strip()
            return self.open_application(app)

        # System info commands
        elif any(word in command for word in ["system", "performance", "cpu", "memory"]):
            return self.get_system_info()

        # Wikipedia commands
        elif any(word in command for word in ["wikipedia", "wiki", "tell me about"]):
            query = command.replace("wikipedia", "").replace("wiki", "").replace("tell me
about", "").strip()
            return self.wikipedia_search(query)

        # Math and calculations
        elif any(word in command for word in ["calculate", "math", "compute", "what is"]):
            if self.wolfram_client:
                return self.wolfram_query(command)
            else:
                # Use Gemini for calculations
                prompt = f"Calculate or solve this math problem: {command}"
                return self.get_gemini_response(prompt)
```

```
        # Screenshot and visual tasks
        elif any(word in command for word in ["screenshot", "screen", "see", "visual"]):
            screenshot = self.capture_screenshot()
            prompt = f"Analyze this screenshot and tell me what you see: {command}"
            return self.get_gemini_response(prompt, screenshot)

        # System control commands
        elif any(word in command for word in ["shutdown", "restart", "sleep", "lock"]):
            return self.control_system(command)

        # Stop/exit commands
        elif any(word in command for word in ["stop", "exit", "quit", "goodbye"]):
            self.running = False
            return "Goodbye! Shutting down Jarvis."

        # General AI conversation
        else:
            prompt = f"You are Jarvis, an AI assistant. Respond to this request naturally and
helpfully: {command}"
            return self.get_gemini_response(prompt)

    def continuous_listening(self):
        """Continuous listening mode"""
        while self.running:
            try:
                command = self.listen()
                if command:
                    response = self.process_command(command)
                    if response:
                        self.speak(response)

                # Small delay to prevent excessive CPU usage
                time.sleep(0.1)

            except KeyboardInterrupt:
                self.speak("Jarvis shutting down")
                self.running = False
                break
            except Exception as e:
                print(f"Error in continuous listening: {e}")
                time.sleep(1)
```

```
    def run(self):
        """Main run method"""
        self.speak("Jarvis AI is now online. How can I help you?")
        print("\n" + "="*50)
        print("JARVIS AI VOICE ASSISTANT")
        print("="*50)
        print("Commands you can try:")
        print("- 'Jarvis, what time is it?'")
        print("- 'Jarvis, search for Python tutorials'")
        print("- 'Jarvis, open calculator'")
        print("- 'Jarvis, what's the weather?'")
        print("- 'Jarvis, tell me about artificial intelligence'")
        print("- 'Jarvis, what is 25 times 47?'")
        print("- 'Jarvis, take a screenshot'")
        print("- 'Jarvis, system performance'")
        print("- 'Jarvis, goodbye' (to exit)")
        print("="*50 + "\n")

        # Start continuous listening
        self.continuous_listening()

def main():
    # Configuration
    GEMINI_API_KEY = "YOUR_GEMINI_API_KEY_HERE"  # Replace with your Gemini API key
    WOLFRAM_API_KEY = "YOUR_WOLFRAM_API_KEY_HERE"  # Optional: Replace with
Wolfram Alpha API key

    # Validate API key
    if GEMINI_API_KEY == "YOUR_GEMINI_API_KEY_HERE":
        print("ERROR: Please replace 'YOUR_GEMINI_API_KEY_HERE' with your actual Gemini
API key")
        print("Get your API key from: https://makersuite.google.com/app/apikey")
        return

    try:
        # Initialize and run Jarvis
        jarvis = JarvisAI(GEMINI_API_KEY, WOLFRAM_API_KEY)
        jarvis.run()

    except Exception as e:
        print(f"Failed to initialize Jarvis: {e}")
```

```
        print("Make sure you have all required dependencies installed:")
        print("pip install speechrecognition pyttsx3 google-generativeai requests wikipedia wolframalpha pyautogui psutil opencv-python pillow")

if __name__ == "__main__":
    main()
```

30 Python Interview Questions

🟢 BASIC LEVEL (Questions 1-10)

1. Data Types and Variables

What are the main data types in Python? How do you check the type of a variable at runtime?

2. Mutable vs Immutable

Explain the difference between mutable and immutable objects in Python. Give examples of each.

3. List vs Tuple

What are the key differences between lists and tuples? When would you use one over the other?

4. String Operations

How do you reverse a string in Python? Show at least two different methods.

5. Dictionary Basics

How do you merge two dictionaries in Python? What happens if they have common keys?

6. Exception Handling

What is the difference between `try-except` and `try-except-finally`? When would you use each?

7. Function Arguments

Explain the difference between `*args` and `**kwargs`. Provide an example of when you'd use each.

8. List Comprehensions

Convert this for loop into a list comprehension:

result = []

```
for i in range(10):
  if i % 2 == 0:
    result.append(i**2)
```

9. Boolean Operations

What is the output of the following expressions and why?

- `bool([])`
- `bool([0])`
- `bool("")`
- `bool("0")`

10. Variable Scope

What will be the output of this code and why?

```
x = 10
def func():
  x = 20
  print(x)
func()
print(x)
```

🟡 INTERMEDIATE LEVEL (Questions 11-20)

11. Decorators

What are decorators in Python? Write a simple decorator that measures the execution time of a function.

12. Generators

What is the difference between a generator and a regular function? How do you create a generator?

13. Class and Objects

Explain the concepts of class, object, and instance in Python. What is the `__init__` method?

14. Inheritance

What is method resolution order (MRO) in Python? How does multiple inheritance work?

15. File Handling

What's the difference between opening a file with `open()` and using `with open()`? Why is the latter preferred?

16. Lambda Functions

What are lambda functions? When should you use them and when should you avoid them?

17. Memory Management

How does Python handle memory management? What is garbage collection and when does it occur?

18. Global Interpreter Lock (GIL)

What is the Global Interpreter Lock in Python? How does it affect multithreading?

19. Deep vs Shallow Copy

Explain the difference between deep copy and shallow copy. When would you use each?

20. Iterator Protocol

What makes an object iterable in Python? How do you implement the iterator protocol in a custom class?

🔴 ADVANCED LEVEL (Questions 21-30)

21. Context Managers

What are context managers? How do you create a custom context manager using both class-based and function-based approaches?

22. Metaclasses

What are metaclasses in Python? Provide a simple example of when you might use them.

23. Monkey Patching

What is monkey patching? Show an example and discuss when it might be useful or dangerous.

24. Multithreading vs Multiprocessing

When would you use threading vs multiprocessing in Python? What are the trade-offs of each approach?

25. Database Operations

How would you prevent SQL injection when working with databases in Python? Show an example with proper parameter binding.

26. Performance Optimization

What are some common techniques for optimizing Python code performance? Mention at least 5 different approaches.

27. Design Patterns

Implement the Singleton design pattern in Python. What are the pros and cons of using it?

28. Error Handling Best Practices

What are the best practices for exception handling in Python? How do you create custom exceptions?

29. Code Testing

What is the difference between unit testing, integration testing, and functional testing? How do you write unit tests in Python?

30. Web Framework Knowledge

Compare Flask and Django. What are the key differences and when would you choose one over the other?

📋 Question Categories Breakdown

Language Fundamentals (40%)

- Data types and structures
- Functions and scope
- Object-oriented programming
- Error handling

Advanced Features (30%)

- Decorators and generators
- Context managers
- Memory management
- Concurrency

Best Practices (20%)

- Code optimization
- Testing approaches
- Design patterns
- Security considerations

Framework/Library Knowledge (10%)

- Web frameworks
- Database interactions
- Third-party libraries
- Development tools

🎯 Interview Preparation Tips

For Basic Questions:

- Practice coding simple algorithms
- Understand Python syntax deeply
- Know built-in functions and methods
- Be able to explain concepts clearly

For Intermediate Questions:

- Implement advanced features from scratch
- Understand Python internals
- Practice object-oriented design
- Know when to use different approaches

For Advanced Questions:

- Study Python's implementation details
- Understand performance implications
- Know design patterns and best practices
- Be familiar with popular frameworks

General Interview Strategy:

1. **Think out loud** - explain your reasoning
2. **Ask clarifying questions** - understand requirements
3. **Start simple** - then add complexity
4. **Consider edge cases** - show thorough thinking
5. **Discuss trade-offs** - show decision-making skills

⏰ Recommended Study Schedule

Week 1-2: Basic Questions (1-10)

- Review Python fundamentals
- Practice basic coding problems
- Understand data structures

Week 3-4: Intermediate Questions (11-20)

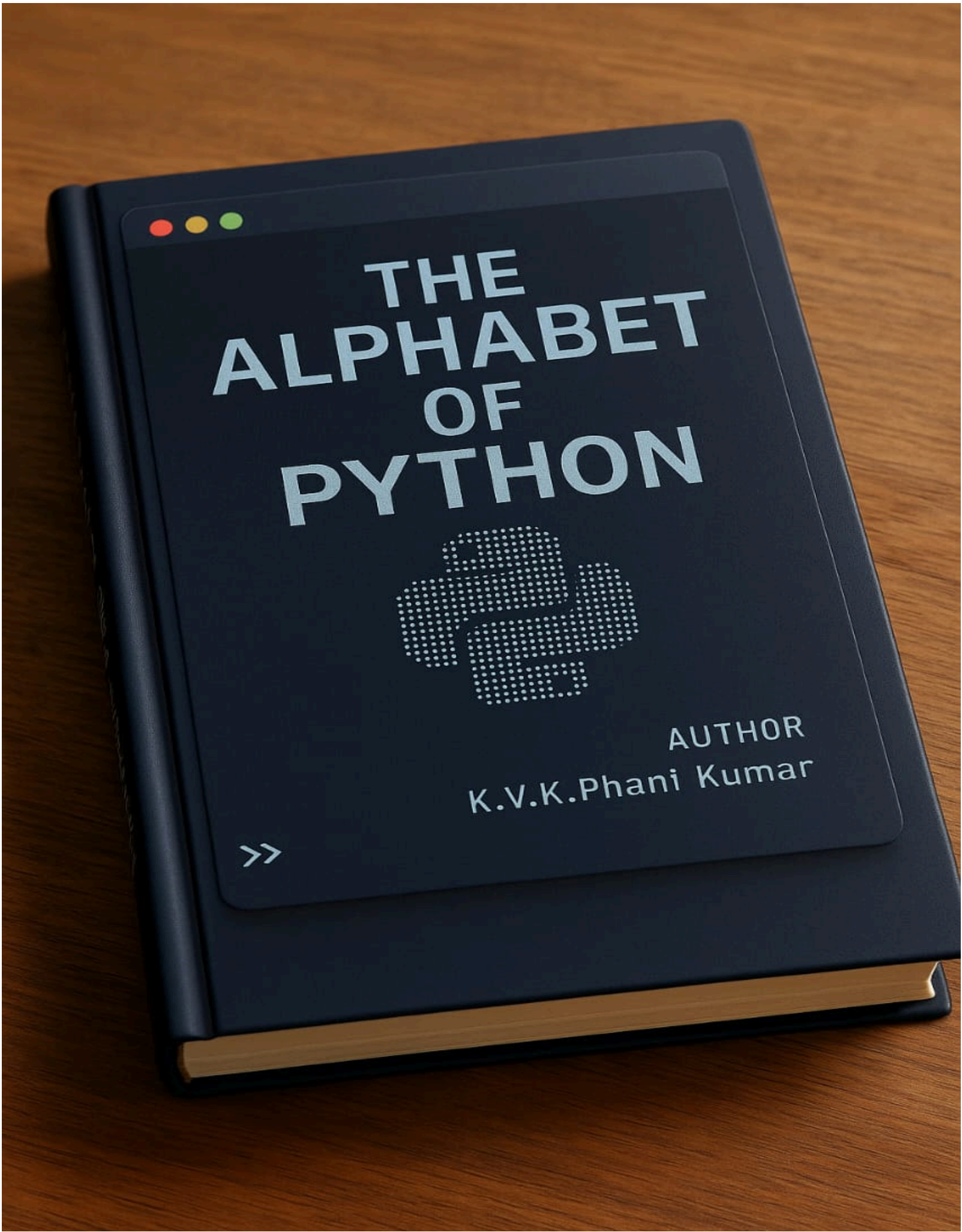
- Study advanced Python features
- Practice OOP concepts
- Learn about Python internals

Week 5-6: Advanced Questions (21-30)

- Explore design patterns
- Study performance optimization
- Learn framework comparisons

Practice Tips:

- Code solutions by hand first
- Explain concepts to others
- Time yourself on coding questions
- Review Python documentation regularly


THE
ALPHABET
OF
PYTHON
AUTHOR
K.V.K.Phani Kumar